Plataforma de simulación reconfigurable basada en Microsoft Robotics Developer Studio

Proyecto de Sistemas Informáticos,
Facultad de Informática,
Universidad Complutense

2012/2013

Director:

Jose Antonio López Orozco

Autores:

Francisco Alcalá Tomás

Ana Celorio Aponte

Cristina Montoya Álvarez

Autorización de difusión

Con este documento, los autores Francisco Alcalá Tomás, Ana Celorio Aponte y Cristina Montoya Álvarez, autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado.

Agradecimientos

Queremos dar nuestro más sincero agradecimiento a nuestra familia, por toda la paciencia y el apoyo recibido a lo largo de estos meses y de nuestra vida universitaria.

También a nuestro director del proyecto, José Antonio López Orozco, por todo el tiempo que ha invertido en nosotros y por su paciencia en los momentos duros.

A Natalia, Ernesto y Álvaro, por tener que soportarnos en el momento más duro de nuestra experiencia universitaria.

Sin ninguno de vosotros no habría sido posible llegar hasta aquí.

A todos ellos, muchas gracias.

Índice

Índice de figuras

Resumen/Abstract

En el proyecto explicado a continuación, se ha diseñado un sistema de módulos reconfigurables que permite realizar simulaciones de distintos tipos de sistemas. Incluye una descripción detallada de la herramienta principal utilizada, el Microsoft Robotics Developer Studio(MRDS), explicando cada una de sus partes, principalmente DSS y CCR, así como la manera de programar con ella. Se han descrito una serie de ejemplos con distintos tipos de sistemas, discretos, continuos y con eventos, donde se prueba la eficacia del diseño de módulos reconfigurables que se ha implementado y las simples modificaciones que se deben realizar para crear uno partiendo de otro ya creado. Incluye un ejemplo con una simulación, donde se explica la parte del simulador de dicha herramienta. Se realiza una demostración de la creación de aplicaciones distribuidas, que permiten ejecutar los distintos servicios en nodos situados en equipos diferentes. Finaliza con las conclusiones que se han obtenido tras la realización del proyecto, todas positivas ante la eficacia y facilidad de uso del simulador siguiendo el sistema de módulos diseñados.

In the project explained below, we have designed a reconfigurable modular system that allows simulations of different types of systems. Includes a detailed description of the main tool used, the Microsoft Robotics Developer Studio, explaining each of its parts, mainly DSS and CCR, as well as how to program with it. We have described a number of examples with different types of systems, discrete, continuous and with events, which proves the effectiveness of reconfigurable modular design that has been implemented and the simple modifications that should be made to create one based on another already created. It includes an example of a simulation, which includes an explanation of the simulator of this tool. We performed a demonstration of creating distributed applications, which allow you to run various services on nodes located on different computers. It ends with the conclusions that have been obtained after the completion of the project, all positive after testing the effectiveness and ease of use of the simulator following the modules system designed.

Palabras clave:

Aplicaciones distribuidas, CCR, DSS, Microsoft Robotics (MRDS), Servicios, Simulación.

Keywords:

Distributed applications, CCR, DSS, Microsoft Robotics (MRDS), Services, Simulation.

1. Introducción

1.1 Objetivos

En este proyecto se persigue el diseño y construcción de una plataforma reconfigurable que permita realizar simulaciones de diferentes sistemas. Es decir, buscamos una plataforma de simulación de sistemas físicos versátil, genérica y reconfigurable de modo que el usuario no necesite un alto conocimiento del entorno que se vaya a utilizar sino que aprovechándose de sus propiedades sólo tenga que centrarse en la definición de las propiedades físicas del sistema a simular, y su interacción con otros elementos existentes. De modo que la distribución de los distintos elementos, su representación en el simulador y su ejecución en paralelo sea transparente al diseño realizado.

Esta estructura quedará definida en una serie de plantillas, elementos, clases u otro tipo de estructuras genéricas que abstraiga al diseñador de todos los elementos de comunicación y de la infraestructura necesaria para realizar la simulación.

Buscamos un sistema de módulos independientes que siga aproximadamente el siguiente esquema:

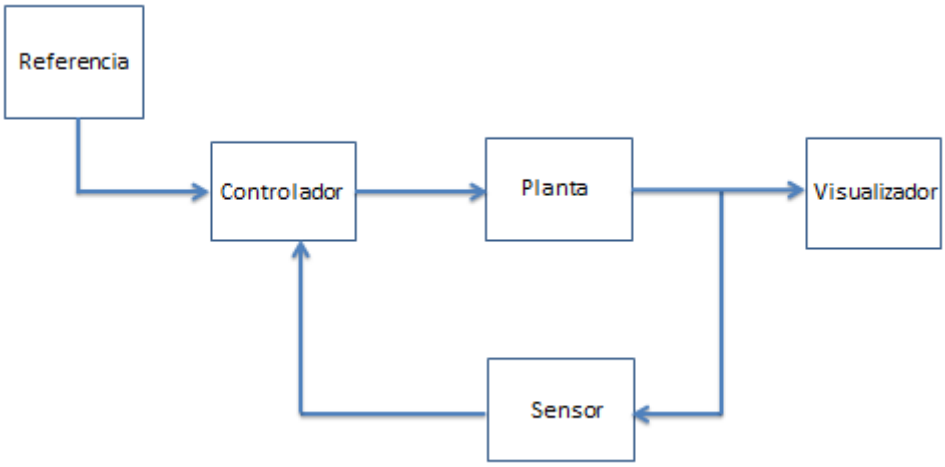


Figura 1.1 – Esquema general de un sistema de control

De este modo, el diseñador del sistema a simular podrá centrarse en el contenido del modelo (sus ecuaciones, comportamiento, etc.) que se definirá por medio de un programa en uno de los módulos definidos. Después puede crear un *Controlador* adecuado partiendo de uno existente, pero mientras, si lo desea podrá hacer uso de los elementos *Sensores*, *Referencia* y *Visualizador* genéricos que se ofertan con la plataforma y posteriormente definir unos propios a partir de los existentes. Esto

permite que la simulación sea más modular, centrándose en el funcionamiento correcto del modelo en lugar de dedicar demasiado esfuerzo a elementos ajenos a él como son la visualización de resultados, la creación de threads, la comunicación entre procesos, etc.

Con este proyecto también se busca facilitar la documentación necesaria para la realización de proyectos de simulación de sistemas como los que se van a estudiar, y toda la información necesaria para el uso de la herramienta MRDS, para que en un futuro, personas que vengas con objetivos similares puedan apoyarse en él, y evitar así mucho tiempo y esfuerzo del que hayamos invertido nosotros. [1]

1.2 Estado del arte: Herramientas de Simulación

Vamos a realizar una comparación de los posibles simuladores[2] que podemos utilizar según cada uno de los intereses que tenemos para el proyecto y lo que nos ofrece cada uno de ellos. Buscamos un simulador con código abierto y con posibilidad de crear aplicaciones distribuidas para poder ejecutar la aplicación desde distintos ordenadores. Nos interesa poder realizar una aplicación reconfigurable y con facilidad de crear módulos independientes. Necesitamos que el simulador que escojamos funcione con un motor de física potente.

Empezaremos estudiando los distintos motores de física, el software capaz de realizar simulaciones de ciertos sistemas físicos como la dinámica del cuerpo rígido, el movimiento de un fluido y la elasticidad. Para ello, hacemos uso del SDK de la física, el encargado de simular los movimientos de cuerpos según leyes de la física.

Los simuladores físicos trabajan según la detección de colisiones. Estos difieren en la forma en que reaccionan en una colisión. Suelen funcionar de dos maneras, en donde se detecta la colisión a posteriori o a priori. La detección de colisiones se refiere al problema de cálculo de la detección de la intersección de dos o más objetos.

Como posibles motores de física, encontramos **Open Dynamics Engine (ODE),** una biblioteca de código abierto, de alto rendimiento para la simulación dinámica de cuerpos rígidos. Está totalmente equipado, estable y con una plataforma C / C + + fácil de utilizar.

Tiene algunos métodos muy útiles, como por ejemplo el método de aproximación de fricción. Encontrando como ventajas que es gratuito y de código abierto.

Otra opción es el **Cafu Engine**, es un motor multiplataforma y actualmente arranca bajo Windows y Linux. Como ventajas encontramos que está disponible como software libre bajo la licencia de GNU y es gratuito. Escrito en C + +, las herramientas,

las bibliotecas y los marcos se han diseñado para hacer fácil el desarrollo de nuevos juegos y aplicaciones 3D.

Encontramos también, **AGX Multiphysics**, es un motor de física para aplicaciones profesionales e industriales. Se desarrolló para satisfacer tanto la estabilidad y el funcionamiento para simuladores de realidad virtual y efectos visuales, así como los requisitos sobre el realismo y la precisión en aplicaciones. Simula la dinámica de cuerpos rígidos, detección de colisiones, contactos de fricción, sistemas articulados, motores, líquidos, materiales deformables, líneas y cables. Arranca bajo Windows, Linux y Mac OS. Pero tiene el inconveniente de ser de pago.

Un motor de física muy potente, es el **NVidia PhysX**. **PhysX** es un motor propietario de capa de software intermedia o "middleware" y un kit de desarrollo diseñados para llevar a cabo cálculos físicos muy complejos. Los motores físicos de middleware permiten a los desarrolladores de videojuegos abstraerse durante el desarrollo, ya que PhysX proporciona funciones especializadas en simulaciones físicas complejas, lo cual da como resultado una alta productividad en la escritura de código.

PhysX es capaz de crear efectos de física dinámica tan impresionantes como las explosiones, la interacción con los escombros y desechos o el movimiento natural del agua y los personajes sin perder rendimiento. En general, aumenta el rendimiento de todas las aplicaciones gráficas. Y se puede obtener de manera gratuita.

Otro motor de física potente, **JigLibX PhysX Library,** es un motor avanzado de física,y es gratuito escrito en C# utilizando el framework de Microsoft XNA. Está basado en el motor de física JigLib y actualmente está siendo portado y ampliado. Tener un sistema de colisión y un motor de física de cuerpo rígido hace que JigLibX sea uno de los motores libres de física de código abierto favoritos.

Una vez estudiado ventajas e inconvenientes, como el precio, si son o no código abierto y lo potentes que pueden ser los motores de física que encontramos en el mercado y hemos estudiado anteriormente, procedemos a la búsqueda de un simulador. Empezaremos con los simuladores que trabajan con alguno de los motores descritos arriba, como puede ser, **Microsoft Robotics Developer Studio (MRDS**), es un entorno basado en Windows para el control robótico y la simulación. Es de libre disposición basada en entorno de programación para crear aplicaciones de robótica. Permite a los aficionados y desarrolladores profesionales o no profesionales crear aplicaciones de robótica dirigidas a una amplia gama de escenarios. Admite soporte para el sensor Kinect y MARK (robot móvil autónomo con Kinect). Microsoft Robotics Developer Studio puede soportar una amplia gama de plataformas robóticas, ya sea corriendo directamente en la plataforma (si se tiene un PC integrado con Windows) o controlar el robot desde un PC con Windows a través de un canal de comunicación. Además de proporcionar apoyo para Microsoft Visual Studio 2010,

Microsoft Robotics Developer Studio 4 ofrece un Lenguaje de Programación Visual (VPL), que permite a los desarrolladores crear aplicaciones simplemente arrastrando y soltando componentes en un lienzo y el cableado entre sí. El MRDS consta de varios componentes, el CCR (Concurrency and Coordination Runtime) es una biblioteca gestionada que proporciona clases y métodos para gestionar con la concurrencia, coordinación y control de errores. El CCR hace que sea posible escribir los segmentos de código que operan independientemente. Y el DSS (Decentralized Software Services) una biblioteca que extiende el concepto de CCR a través de procesos e incluso a través de máquinas. Una aplicación creada con el DSS se compone de varios servicios independientes que se ejecutan en paralelo.

Robotics Developer Studio es gratuito, de código abierto y requiere Windows 7. Además de utilizar Robotics Developer Studio como un entorno de desarrollo independiente, puede ser utilizado con cualquiera de los Visual Studio con C# para crear servicios independientes con distintas funcionalidades. Tiene además un mecanismo muy sencillo para crear aplicaciones distribuidas, cumpliendo así con todos los objetivos planteados.

Por otro lado, encontramos el simulador Microsoft XNA Game Studio, la arquitectura utilizada por Xbox, que ofrece un conjunto de herramientas con un entorno de ejecución administrado proporcionado por Microsoft, que facilita el desarrollo de juegos de ordenador y de la gestión de XNA. Incluye un amplio conjunto de bibliotecas, específicas para el desarrollo de juegos, para promover la máxima reutilización de código a través de plataformas de destino. XNA Game Studio es un entorno de desarrollo integrado que se ha diseñado para facilitar el desarrollo de juegos para Microsoft Windows, Xbox 360 y Windows Phone. Es una biblioteca de clases de código administrado que contiene características destinadas específicamente al desarrollo de juegos. Además, XNA Game Studio incluye herramientas para agregar contenido gráfico y de audio al juego. Para ejecutar juegos XNA Framework en un equipo con un sistema operativo Windows, se necesita una tarjeta gráfica compatible, como mínimo, Shader Model 1.1 y DirectX 9.0c. Es gratuito, y se programa con C#, con el visual Studio 2010. Permite que los desarrolladores de juegos se concentren más en el contenido y la experiencia de juego. Aunque como inconveniente encontramos que no dispone de un método sencillo para la realización de aplicaciones distribuidas.

Tras estudiar las propiedades de ambos, concluimos que por ser Microsoft Robotics Developer Studio el que más propiedades cumple de las buscadas en los objetivos pues es gratuito, de código abierto y facilita la creación de aplicaciones distribuidas, además de utilizar un motor de física muy potente. Así pues, utilizaremos este para la realización del proyecto.

Otros simuladores que podemos encontrar en el mercado, pero no son tan interesantes para este proyecto pues se basan más especialmente en la física, son el

Box2D una biblioteca libre que implementa un motor físico en dos dimensiones. Box2D utiliza MKS (metros, kilogramos, y segundos) y unidades de radianes de ángulos. Box2D portátil está escrito en C + +.

Y **Bullet,** una biblioteca para la gestión de colisiones. La biblioteca ha sido usada en multitud de producciones cinematográficas así como videojuegos. La Biblioteca de Física Bullet es gratuita para uso comercial y de código abierto en C++. Bullet 3D Game Library Multiphysics proporciona el estado de la detección de colisiones y dinámica de cuerpos rígidos.

Como los estudiados hasta ahora son simuladores orientados a la robótica, vamos a estudiar algunos otros simuladores de sistemas. Vemos una serie de simuladores que se encuentran en el mercado, como apoyo para ayudarnos con el uso de nuestro simulador o para orientarnos en algunos casos no orientados a robots. Algunas de las herramientas a las que nos referimos son por ejemplo **MATLAB /Simulink.** Simulink es un entorno de diagramas de bloques para la simulación multidominio y el diseño basado en modelos. Admite el diseño y la simulación a nivel de sistema, la generación automática de código y la prueba y verificación continua de los sistemas embebidos. Simulink ofrece un editor gráfico, bibliotecas de bloques personalizables y solvers para modelar y simular sistemas dinámicos. Se integra con MATLAB, lo que permite incorporar algoritmos de MATLAB en los modelos y exportar los resultados de la simulación a MATLAB para llevar a cabo más análisis. Pero al no estar orientado a la robótica no nos interesa utilizarlo como simulador, pues no es suficiente para lo que buscamos.

También nos puede resultar interesante **Seamless 3D**, es software de código abierto, de modelado 3D y gratuito. Seamless3d se puede utilizar como un editor de malla y un animador, sin embargo, su característica principal es su tecnología de generación de nodo. Construir nodos permite al usuario realizar una secuencia de operaciones complejas en tiempo real cada vez que un punto de control en la ventana 3D es arrastrado. Así,podemos ayudarnos de esta herramienta a la hora de crear mallas o la creación de distintos servicios.

Otra herramienta bastante útil que podemos encontrar es **VisSim**. Es un lenguaje de diagrama de bloques para la creación de complejos sistemas dinámicos no lineales. Para crear un modelo, basta con arrastrar los bloques en el área de trabajo y conectarlos con cable. VisSim motor de matemáticas ejecuta el diagrama directamente sin retardo de compilación.

Otra posibilidad, **SimApp** es un software de simulación dinámica para modelar sistemas en el dominio del tiempo y frecuencia. SimApp ofrece una potencia de simulación significativa a un precio razonable, al tiempo que reduce el tiempo de aprendizaje. SimApp contiene muchas características que le ayudan a hacer crecer el

modelo de sofisticación al nivel necesario para obtener resultados realistas. SimApp también puede ayudarle a ocultar la complejidad del modelo mediante la construcción de elementos de uso común o subsistemas en bloques personalizados.

Una herramienta también interesante**,** **AnyLogic** es la única herramienta de simulación que soporta todas las metodologías de simulación más comunes: Dinámica de Sistemas , centradas en los procesos(Eventos Discretos AKA), y Basada en Agentes de modelado. Incluye un lenguaje de modelado gráfico y también permite que los usuarios puedan ampliar los modelos de simulación con código de Java. Usando los modelos de Java, en AnyLogic se prestan a ser modificados para requisitos particulares y además, se pueden crear applets de Java que se puede ejecutar en cualquier navegador estándar. Estos applets permiten compartir fácilmente los modelos AnyLogic por correo o localizándolos en la red en cualquier website. Además de los applets de Java la versión profesional permite la creación y distribución de aplicaciones completas.

1.3 ¿Qué es Microsoft Robotics Developer Studio?

Lo primero que tenemos que saber, es ver en qué consiste la herramienta con la que vamos a trabajar. En nuestro proyecto, todo se va a basar en torno a Microsoft Robotics Developer Studio[3], cuyas iniciales son MRDS y son las que vamos a utilizar para referirnos a él.

MRDS es una herramienta para la creación de aplicaciones robóticas, está orientado al ámbito académico y a los aficionados al mundo de la robótica ya que es más accesible en comparación con el resto, además de ser gratuito. Es algo costoso de conocer su funcionamiento cuando comencemos a utilizar dicha herramienta, pero después es muy sencillo realizar implementaciones con ella. El modelo de arquitectura se basa en la estructura Cliente-Servidor. Para entenderlo de una manera mejor, cuando el programador ha creado un proyecto de MRDS, crea un servicio, y este servicio es ejecutado por un cliente. En este caso, el cliente es una Máquina Virtual llamada **DSS** y que será explicada más adelante con más detalle.

Posee un soporte de tiempo real orientado a servicios y está desarrollado sobre el entorno .NET y soporta lenguajes de programación como VB.NET, Python , Visual Basic, VPL o C#. Nuestro proyecto está realizado con C# ya que está centrado en el manejo de comunicación entre los servicios y es el más apropiado para hacerlo. Una vez decidido cual será nuestro lenguaje de programación, debemos saber que el resultado de compilar nuestra aplicación será un archivo XML que lo llamaremos **manifiesto**. Dicho manifiesto es el que describe a los servicios y será ejecutado por el **DSS** (que es nuestra máquina virtual) ya que es a través de donde se comunica con otros servicios.

Entre las ventajas que posee, podemos destacar que es una plataforma extensible, escalable y que la podemos utilizar tanto para un entorno simulado como un entorno real conectando el robot por un puerto serie del PC, Bluetooth o Wifi.

Debemos saber que MRDS consta de tres componentes principales: entorno de simulación (VSE), el lenguaje (VPL) y el entorno de ejecución (CCR y DSS).

VSE está diseñado para utilizar una variedad de escenarios. La integración de AGEIA PhysX permite realizar una simulación física con una evolución constante.

VPL está enfocado a usuarios poco experimentados y que necesitan adaptarse rápidamente a la plataforma ya que es muy sencillo y gráfico.

Por último, el entorno de ejecución se compone de CCR (Concurrency and Coordination Runtime) y DSS (Decentralized Software Services) que pasarán a explicarse con mayor detalle a continuación.

En la siguiente imagen se observa el esquema general del programa MRDS.

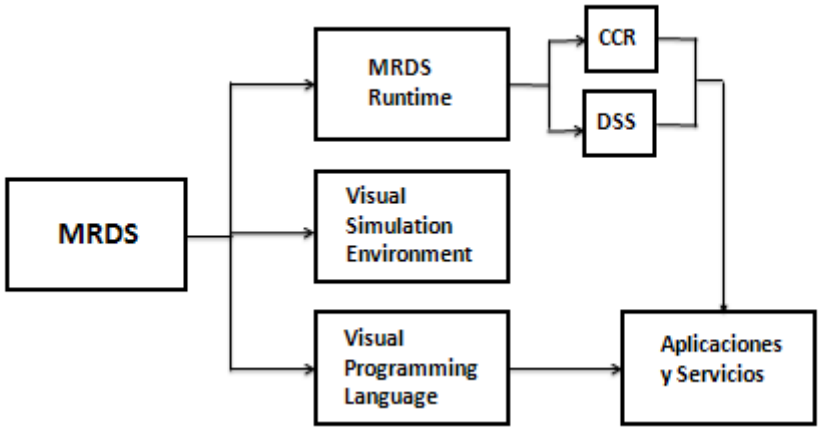


Figura 1.3 -1 Esquema Microsoft Visual Studio 2010

1.3.1 Microsoft Visual Estudio

Microsoft Visual Estudio es un entorno de desarrollo integrado para sistemas operativos Windows. Con él, podemos programar en varios lenguajes de programación como C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET. Debido a que C# es el lenguaje de programación más adecuado para la implementación de nuestro proyecto, hemos utilizado dicha herramienta que es la que más nos facilita su desarrollo.

Con Visual Studio admitimos numerosas herramientas que hacen la implementación en C# mucho más sencilla que con cualquier otro editor. Posee un editor de código completo, plantillas de proyecto, asistentes para código, un depurador de fácil manejo y eficaz, además de otras muchas herramientas. También posee una biblioteca de clases .NET Framework que ofrece acceso a un número elevado de servicios y a otras clases que pueden resultar útiles y que pueden resolvernos numerosos problemas.

Al combinar .NET Framework con C#, podemos crear aplicaciones para Windows, servicios Web, herramientas para bases de datos y mucho más.

1.3.2 MRDS Runtime

MRDS Runtime proviene de un entorno de ejecución (servidor) que será quien se encargue de crear, administrar y conectar los diferentes servicios que forman parte de una aplicación robótica. Para ello, el entorno de ejecución hace uso de 3 niveles:

- Decentralized Software Services (DSS).
- Concurrency and Coordination Runtime (CCR).
- Common Language Runtime (CLR).

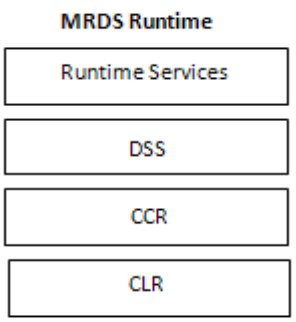


Figura 1.3.2-1 MRDS Runtime

1.3.2.1 Decentralized Software Services (DSS)

Lo primero que podemos decir acerca del DSS, es que es el máximo responsable de controlar aplicaciones de robótica, así como de iniciar y detener servicios o comunicarlos entre ellos administrando el flujo de mensajes. Está situado en un nivel por encima del CCR, como se puede ver en la figura del MRDS Runtime (Figura 1.3.2-1), y proporciona un modelo orientado a servicios.

Está basado en una arquitectura REST (Representational State Transfer) de servicios web que permite que los dispositivos se puedan comunicar en un entorno distribuido desde el principio. La dinámica de la arquitectura REST de los servicios, está basada en los cambios respecto a su estado interior, y los mensajes y notificaciones que intercambian entre sí que afectan al mismo.

Respecto a los servicios, son los bloques con los que construimos las aplicaciones de MRDS y éstas pueden representar acceso al HW (como la lectura de sensores: bumper, laser, batery…) y SW (servicios para UI, Directory Services, etc). Como dijimos antes, los servicios pueden comunicarse entre sí mediante el paso de mensajes. Una parte importante en el manejo de servicios, son los nodos DSS. Un nodo DSS consiste en un ambiente de ejecución que da soporte a los servicios para ser creados y manejados hasta que se borran o el nodo DSS se para.

Hay que decir que DSSP es un protocolo propio que ofrece DSS encargado de la mensajería entre servicios y la creación de servicios descentralizados. Además permite que el estado se mantenga durante el periodo de vida de la aplicación.

Más adelante explicaremos más detalladamente DSS y los servicios:

Los servicios tienen que seguir el siguiente esquema, véase la figura 1.3.1.1-1:

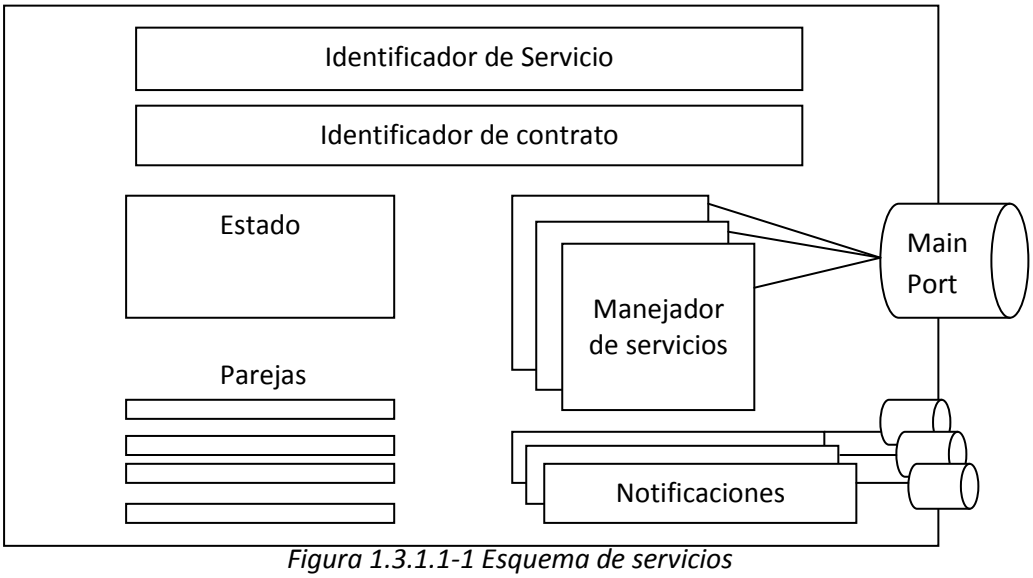


Figura 1.3.1.1-1 Esquema de servicios

- **Service Identifier**: Es la dirección url que representa al servicio.

- **Contract Identifier**: Es el identificador del contrato de operaciones que este servicio acepta. Debemos definir cuáles son las operaciones o puertos que este servicio proporciona.

- **State**: Representa el estado del servicio y es la información que se va a obtener de la operación GET. Todos los servicios tienen una clase Estado, pero la tenemos vacía. Tenemos que añadir campos al estado cuando creamos un Servicio.

- **Main Port**: Es el puerto principal de comunicaciones del servicio. La clase PortSet es el lugar en el que la vamos a empezar a usar. MainPort es un PortSet con las operaciones (DssOperations<TBody, TResponse>, con esto estamos definiendo el tipo de dato de la petición y la respuesta a la operación del servicio). DSSP está basado en REST, que es una tecnología que normalmente usa HTT, que está basado en peticiones y respuestas (Request y Response). Básicamente lo que estamos haciendo aquí es decirle en la operación del protocolo DSSP cuál es el tipo de la petición, ósea que valores necesito para mi petición, y cuál es el tipo de respuesta, que son los valores de respuesta de la operación.

- **Partners**: Son la lista de servicios externos que este servicio necesita para funcionar si es que tiene algún partner.

1.3.2.2 Concurrency and Coordination Runtime (CCR)

El CCR es la pieza que une el MRDS con un servicio. Es un componente DLL (Dynamic Linking Library) accesible desde cualquier lenguaje de programación abarcado por .NET (Es una librería .NET). Se encarga de sincronizar múltiples procesos. Permite ejecutar múltiples tareas simultáneas en un mismo equipo.

DSS combina aplicaciones (servicios) y también permite ejecutarlas en distintas máquinas. Necesitamos esta concurrencia porque, a la vez que podemos conducir un robot necesitamos estar escuchando también a los sensores. Por ejemplo, sin este procesamiento asíncrono, un robot movería solo una rueda a la vez. El robot tendría que esperar a que cada servicio completase lo que estaba haciendo antes de empezar otro servicio. Por lo tanto, se encarga de controlar multitud de dispositivos que funcionan al mismo tiempo.

Permite la coordinación de mensajes, antes teníamos que usar multithreading, semáforos, pero corríamos el riesgo de que se pudiera bloquear, sin embargo ahora

tenemos el mecanismo de threads. Plantea un modelo de programa que facilita las operaciones asíncronas de las aplicaciones explotando el hardware paralelo.

La orquestación, pone los servicios juntos, trata de ordenar los distintos servicios, combinarlos y pasar mensajes entre ellos, de esto se encargara el DSS.

Estructura del CCR:

Pasamos **mensajes** de cualquier tipo por **ports** (**puertos** que son colas que manejan mensajes sólo del mismo tipo). Se coloca el mensaje en un puerto esperando un **receptor**. Se pueden poner **condiciones de activación** en los receptores para hacer cosas más complejas como un join.

La evaluación de esas condiciones la realizan los arbiters **(Arbitros)**.

Cuando se han cumplido, el mensaje pasa a ser task **(tarea)** y pasa a otra cola de un **Dispatcher**(despachador) y luego este lo ejecuta.

Cuando una tarea se va a ejecutar, se ejecuta un **Handler** (manejador, trozo de código que se ejecuta en multithreading, genera nuevos mensajes y continua).

1.3.2.3 Common Language Runtime (CLR)

Se trata de un entorno de ejecución para los códigos de los programas que corren sobre la plataforma Microsoft .NET. El CLR es quien se encarga de compilar una forma de código intermedio llamada CIL (Common Intermediate Language) al código de máquina nativo, mediante un compilador en tiempo de ejecución. No debemos confundir CLR con una máquina virtual, ya que una vez que el código se ha compilado, empieza a ejecutarse sin que sea interrumpido.

Los desarrolladores que usan CLR, escriben el código fuente en un lenguaje compatible con .NET, en nuestro caso utilizamos el ya mencionado C#, pero también se puede utilizar Visual Basic .NET. En tiempo de compilación, el compilador .NET convierte el código recibido a CIL. Y en tiempo de ejecución, el compilar del CLR convierte el código CIL en código nativo para el sistema operativo, figura 1.3.1.3-1.

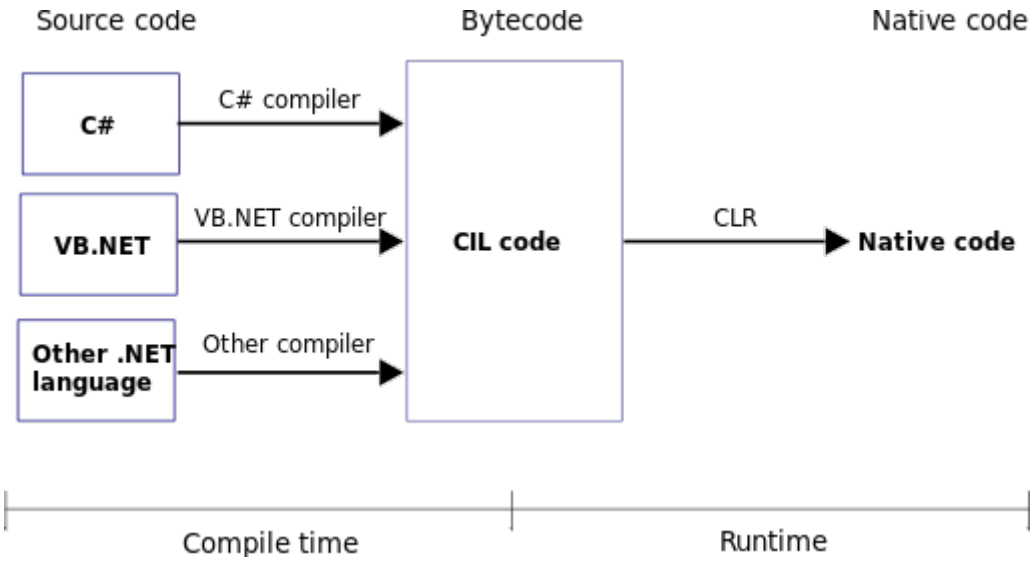


Figura 1.3.1.3 – 1Tracducciones de código fuente a código nativo

1.3.2.4 Visual Simulation Environment (VSE).

MRDS incluye un entorno de simulación completo con simulaciónes físicas, en el cual utilizamos gráficos 3D de alta calidad para realizar un mundo virtual. De esta manera, MRDS ofrece a los programadores una alternativa de acceso al mundo de la robótica sin tener la necesidad de adquirir ningún tipo de Hardware. Algunos de los modelos de robots incluidos son LEGO NXT, iRobot Create, Pioneer 3DX o el brazo robótico KUKA LBR3.

1.3.2.5 Visual Programing Language (VPL).

MRDS también incluye una herramienta para la generación de código de una manera visual, llamada VPL. La pueden usar tanto programadores novatos como expertos. Consiste en arrastrar y colocar una secuencia de conexiones de bloques con entradas y salidas que pueden conectarse a otros bloques, que representan actividades o servicios, dentro de un área de diseño. La aplicación resultante es denominada como orquestación.

VPL ofrece dos tipos de actividades: En primer lugar las actividades básicas, que son las que incluyen entrada de información, cálculos y variables. En segundo lugar, las actividades de servicio, que vienen creadas por el propio MRDS o son creadas por nosotros mismos.

2. Microsoft Robotics Developer Studio

Como ya comentamos en el capítulo anterior, MRDS[4] es la herramienta elegida con la que vamos realizar nuestro proyecto. Vamos a realizar algunas aplicaciones robóticas y también una serie de ejemplos con diferentes sistemas: discreto, continuo y basado en eventos. Sin embargo, en esta sección vamos a entrar en más detalle en el funcionamiento de los dos componentes básicos de MRDS: CCR y DSS.

La principal diferencia que tenemos que tener en cuenta es que el CCR va a seguir un modelo de programación para el manejo multi-threading y la sincronización multitarea. Mientras que el DSS lo usaremos para la construcción de aplicaciones siguiendo un modelo de servicios.

2.1 Concurrency and Coordination Runtime (CCR)

Se trata de una biblioteca de código suministrado, o más bien conocido como DLL (Dynamic Linking Library), al cual podemos acceder desde cualquier lenguaje de .NET. Está diseñado para manejar la comunicación asíncrona entre los servicios que están acoplados y que se están ejecutando al mismo tiempo. Va a permitir al usuario diseñar aplicaciones para que los módulos o componentes de Software, puedan ser acoplados de alguna manera. Esto significa, que lo podemos hacer de una manera independiente y hacer suposiciones mínimas sobre su entorno de ejecución y otros componentes. A continuación mostramos una figura que resume la arquitectura CCR[5].

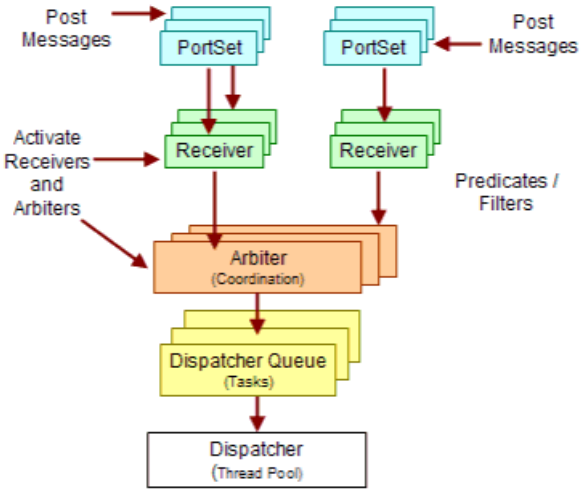


Figura 2.1 -1 Estructura del CCR

- El CCR usa un modelo de programación asíncrona basada en el paso de mensajes utilizando una estructura que denominaremos como *puertos* (ports). Un mensaje es una instancia de cualquier tipo de datos que el CLR puede ser capaz de manejar.
- Los Puertos (Ports) son colas de mensajes que solo aceptan mensajes del mismo tipo ya que así ha sido declarado en la declaración del tipo del puerto. Esto es importante para que lo consideremos como un tipo de seguridad para evitar errores en el tiempo de compilación.
- Colocamos los mensajes en un puerto destinándolos al puerto *Post*. Dichos mensajes permanecen en el puerto hasta que son retirados de la cola por el receptor (*receiver*). Las condiciones de activación se pueden establecer en los receptores para crear expresiones lógicas complejas. La evaluación de dichas condiciones va a ser trabajo de los árbitros (*arbiters*).
- Las primitivas de coordinación, las cuales son aplicadas a través de los árbitros, las utilizamos para sincronizar y controlar tareas. Una vez que las condiciones de un receptor se han cumplido, la tarea se pone en *dispatcher queue* y pasa al *dispatcher* para su ejecución.
- Cuando una tarea se programa para ejecutada, será un *handler* quien se encargue de hacerlo. Los *handler* son partes del código que se ejecutan en un entorno multi-thread. Se pueden ejecutar por su cuenta, si lo único que solicita es la ejecución de una tarea, pero es más habitual que consuma mensajes como consecuencia de estar conectado a un receptor (*receiver).*
- Dentro del CCR existen mecanismo para manejar los fallos. Dichos fallos son denominados *faults*.

El DSS se basa en el modelo del CCR y no requiere de ningún otro componente. Proporciona un entorno de *hosting* para ejecutar servicios y así hacerlos accesibles a través de un navegador web. Es importante tener en cuenta, que el DSS no se puede separar del CCR.

2.1.1 Tareas (Task)

Una tarea contiene una referencia a un controlador que es un trozo de código que se quiere ejecutar. Las tareas implementan ITaskInterface, que proporciona una estructura de datos y permite pasar en cola. Hay que poner cola en ITask antes de pasar al distribuidor (dispatcher). Hay que tener en cuenta el número de hilos y procesos están asignados para no ver que se bloquee ni se ejecute secuencialmente nada que no queramos. Para poder consultar el hilo que ejecuta algo utilizamos Thread.CurrentThread.ManagedThreadId y para modificar la definición de servicio y añadir el atributo, usaremos ActivationSettings. El distribuidor (dispatcher) por defecto del DSS solo tiene dos hilos para el grupo de subprocesos.

Para las Tareas, es necesario activarlas y crear las instancias necesarias. Es importante utilizar la siguiente instrucción, ya que sin ella, no conseguimos activar la tarea y no serviría de nada

Ej: *Arbitrer.Activate(DispatcherQueue , Arbitrer.FromHandler(tarea1), Arbitrer.FromHandler(tarea2)….);*

2.1.2 Ports y PortSet

Un elemento que tienen todos los servicios es el puerto principal. Este puerto implementa una cola FIFO (First-in, First-out) de mensajes. A este puerto llegarán los mensajes que contienen las peticiones que provienen desde otros servicios que a su vez tienen asociados y las respuestas generadas por el servicio en contestación a dichas peticiones. Este mecanismo lo vamos a encontrar implementado en la clase genérica *Port.* Una instancia de esta clase solo podrá recibir mensajes de un determinado tipo, que será declarado durante su instanciación. Un ejemplo de su declaración, sería el siguiente:

```
Port<int> intPort = new Port<int>();
```

Sin embargo, para poder permitir el envío de distintos tipos de mensajes a través de un mismo puerto principal, utilizaríamos la clase PortSet, que implementa un conjunto de colas de distinto tipo y que son tratadas como una única entidad.

De esta manera, podemos definir qué tipo de mensajes se pueden intercambiar entre servicios que estén asociados entre sí en tiempo de compilación.

2.1.3 Arbitrers

CCR nos permite realizar una declaración de *Arbitrers*, que serán quienes nos van a ayudar a realizar el envío y recepción de mensajes entre los diferentes servicios de una manera asíncrona. De esta manera vamos a poder continuar con la ejecución de otras secciones de código mientras se espera la llegada de mensajes a través de algún puerto.

Estos árbitros coordinan una serie de receptores que se van a encargar de observar un puerto especifico a la espera de la llegada de un determinado tipo de mensaje. Tras la llegada de este, el receptor que haya sido definido lo capturará y el árbitro se encargará de decidir qué acciones son las que se realizarán.

En resumen, los árbitros evaluarán los elementos entrantes para decidir que tarea se va a llevar a cabo. Tenemos que tener en cuenta también que existen diferentes tipos de árbitros:

- ***RECEIVE:*** Al declarar un árbitro de tipo receive, es importante tener en cuenta que en los parámetros de la llamada, tiene un indicador que indicará si es

persistente o no, un puerto en el que vamos a recibir la información y un *handler* que lo procesa.

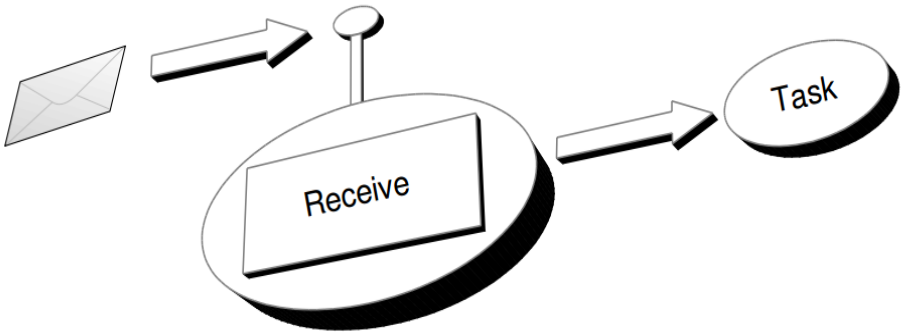


Figura 2.1.2 -1 Receive en Arbitrers

- ***CHOICE:*** Se diferencia del tipo anterior en que tiene dos flujos de ejecución según el contenido del elemento dentro del puerto. Si se trata de un tipo, pasará a ejecutarse una acción, y si es de otro, la otra acción. Por lo tanto, definimos dos tareas en lugar de una. Para un mejor entendimiento, es como si se tratase de un operador OR lógico y se trata principalmente para tratar fallos que podamos encontrar.

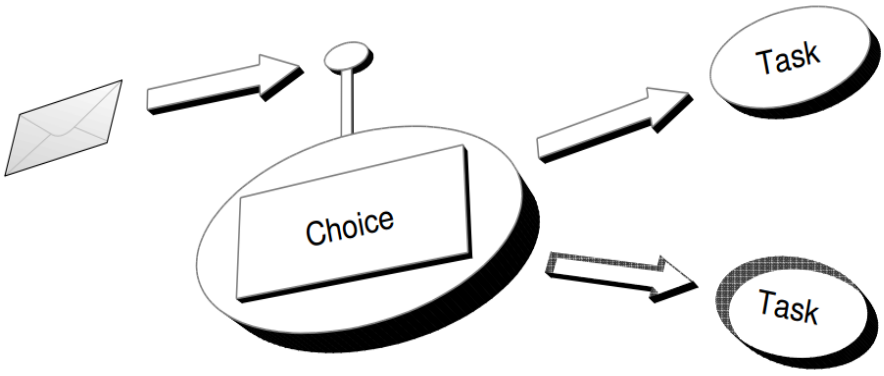


Figura 2.1.2 -2 Choice en Arbitrers

- ***JOIN:*** En este caso, cuando los elementos esperados llegan por dos puertos distintos, entonces es cuando se ejecutará el *handler* y nunca antes. Se traduce como una AND lógica. Por lo tanto, dos procesos acuerdan sincronizarse en un punto de la ejecución de tal manera que hasta que no acaban ambos, no se continúa ejecutando.

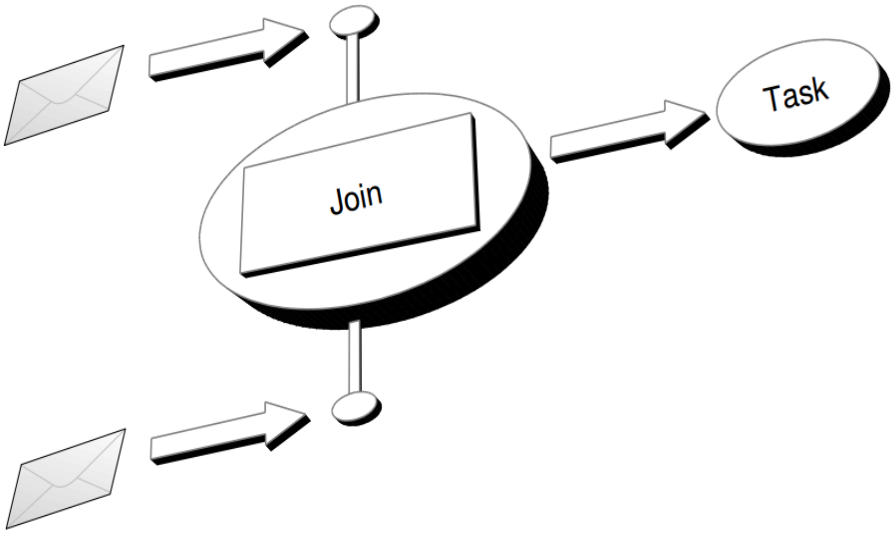


Figura 2.1.2 -3 Join en Arbitrers

Es importante saber que los árbitros se pueden anidar entre sí. Podríamos incluir dentro del árbitro *Choice,* un árbitro *Join* o viceversa. De esta manera podemos jugar con los diferentes funcionamientos de los árbitros con el fin de obtener el resultado deseado.

También existen los siguientes árbitros para situaciones en las que se reciben más de un mensaje:

- ***MultipleItemReceive:*** recibe un nímero concreto de mensajes en solo puerto.

- ***MultiplePortReceive:*** igual que el anterior, la única diferencia es que utiliza un PortSet en lugar de un puerto normal.

También debemos tener en cuenta, el conocimiento de un elemento denominado *Interleave*, que nos permite crear múltiples receptores si así lo deseamos. Podemos diferenciar tres tipos:

- ***TEAR DOWN:*** se ejecuta una sola vez y a continuación, *Interleave* se cierra automáticamente.

- ***CONCURRENTE:*** los procesos se ejecutarán en paralelo dentro de las diferentes posibilidades de las que se dispongan.

- ***EXCLUSIVO:*** se ejecuta de tal manera que bloqueará a los procesos que puedan estar ejecutándose.

2.1.4 Dispatchers (Distribuidores)

Cuando creamos una aplicación en MRDS, es muy importante saber que todas ellas se ejecutan en un entorno multitarea. Por este motivo, tenemos que tener un mecanismo que permita asignar hilos de ejecución a las diferentes tareas que se van a realizar. Todo esto se encarga de realizarlo el CCR, y más concretamente los Distribuidores (Dispatchers). Los distribuidores van a permitir que la ejecución continúe en ciertas partes del código mientras otras están esperando a que ocurra algún evento. CCR va a proporcionarnos una interfaz denominada ITask cuyo objetivo es agrupar las diferentes funciones en una cola de ejecución que van a estar esperando para poder ser ejecutadas.

2.1.5 Time delays

Nosotros estamos familiarizados con System.Threading.Thread.Sleep. Por medio de él, podemos usar Thread.Sleep en un entorno CCR. Sin embargo, está forma de introducir esperas no es la más aconsejada en el CCR, ya que puede llevar a resultados inesperados.

Para introducir esperas o retardos en Robotics Studio, lo más recomendable es usar la clase Timeout[8], es más, podemos usar directamente lo siguiente en un iterador.

```
Yield return Timeout(3000);
```

Siguiendo con lo que hemos comenzado hablando, deberíamos notificar llamadas a Wait que suspenden la ejecución del código temporalmente, simplemente llamara a Thread.Sleep. El "receiver" esperará el time delay especificado en milisegundos. Es peligroso ya que si Wait es llamado mientras por otro Wait, entonces dos threads son bloqueados. Sin embargo, no habrá ningún hilo para ejecutar a los delegates y que despierte a los hilos que han quedado en espera. Solo tendrá el mismo efecto que Thread.Sleep. Así que debemos controlarlo de manera que nuestro programa no se quede en punto muerto.

Podemos tener en cuenta también, que dentro de los iteradores podemos crear delays directamente utilizando TimeoutPort en un yield return:

```
Yield return Arbitrer.Receive(false, TimeoutPort(timeDelay,
delegate (DateTime timeout) {} );
```

Sin embargo, esta opción tampoco es muy segura ya que usa CLR timers.

Otras alternativas que se pueden usar siguiendo la filosofía del CCR (como la mencionada al principio) es la creación de un puerto para indicar que una tarea se ha completado, un puerto Completado. El puerto que utilicemos nos da igual, lo único que nos interesa es que la llegada del mensaje nos notifique que la tarea ha

terminado. Si tenemos varias tareas a las que esperar, podemos hacer que cada una mande un mensaje final al puerto Completado y hacer un MultipleItemsReceive, para esperar que todos hayan terminado.

Y ya por último, podemos usar también la primitiva de CCR executeToCompletion. Esta primitiva nos permite ejecutar el código que indiquemos hasta que termine y después seguir con el resto del código secuencialmente. En resumen, es una llamada que bloquea.

2.1.6 Eventos periódicos

Ejecutar el código periódicamente es una manera de usar TimeoutPort con el su atributo definido como persist puesto a verdadero (true) y así especificar a los manejadores (handler) para la llamada. La aproximación de estos resultados en el manejador (handler), van a ser llamados repetidamente. Pero si el manejador no puede realizar procesos dentro del tiempo especificado, entonces los mensajes retroceden y el handler estará ejecutándose continuamente.

2.2 Decentralized Software Services (DSS)

En este apartado vamos a encargarnos de explicar el funcionamiento del DSS[5] en detalle. Cómo ya comentamos en la sección anterior, DSS va a ser el principal encargado de controlar las aplicaciones de robótica, de crear servicios y que se comuniquen entre ellos mediante el paso de mensajes. A continuación, pasaremos a explicar qué es un servicio.

2.2.1 Servicios

Los servicios[7] son bloques básicos que se utilizan en MSRS para construir aplicaciones. Cada servicio puede combinarse con otros servicios para crear dichas aplicaciones, y serán denominados como ***partners***. A este proceso de combinación lo conoceremos como ***orchestration.*** Todos los servicios están basados en un modelo de aplicación DSS (Manejo de servicios con los nodos DSS). Un nodo DSS es un entorno que proporciona soporte a los servicios (su creación y su manejo), hasta que son eliminados o hasta que el DSS para. Es el encargado de la comunicación entre servicios.

Para entender mejor como funciona un servicio por dentro, tenemos que seguir el siguiente esquema:

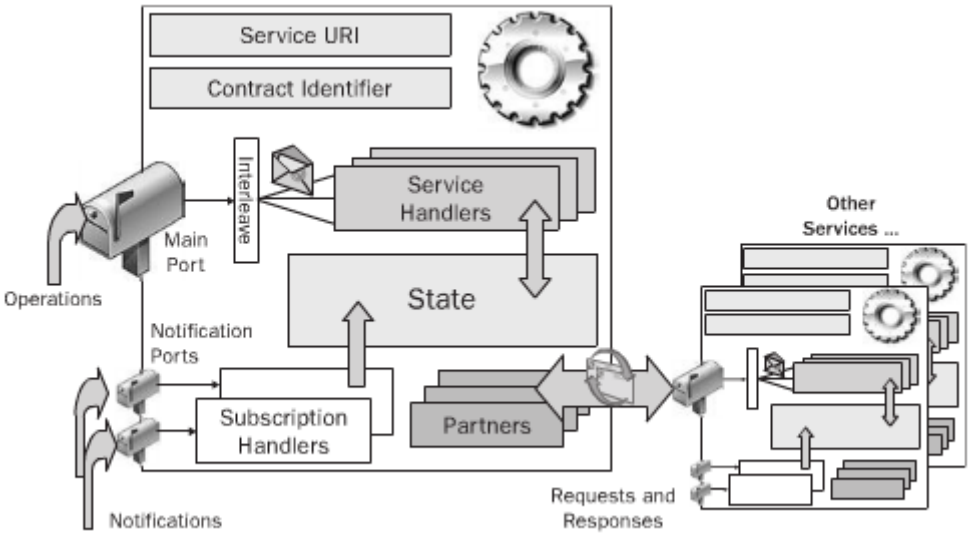


Figura 2.2.1-1 Esquema del DSS

- **Service URI**: Es la dirección URL que representa al servicio.

- **Contrato (Contract Identifier):** Describe el comportamiento del servicio: Descripción de las distintas operaciones que realiza el servicio, y de los datos para ejecutar cada una de estas operaciones.

- **Estado (State):** Información referente al servicio. Accesible por cualquier otro servicio con el que se haya creado una asociación. Si queremos que alguna información de nuestro servicio sea accedida por otros, hay que añadirla aquí. Esta información se obtiene a través de la operación GET.

- **Socios (Partners):** (Subscribe) Los servicios interactúan con otros mediante la creación de asociaciones. Permiten acceder a las operaciones ofrecidas por los servicios, así como recibir información procedente de eventos que se han producido en los socios mediante mensajes de tipo modificación. (Las asociaciones son bidireccionales).

- **Puertos (Ports):** Sistemas de comunicación básicos utilizados por los servicios para el envío y recepción de mensajes referentes a eventos que se producen en los servicios. Existen dos tipos de puerto: Principal y Notificación. El principal se encarga de los mensajes de tipo interno (para el servicio mismo), y el de notificación es para crear asociaciones con otros servicios, permitiendo así la comunicación. *Los de notificación hay que suscribirlos al puerto principal.

- **Manejadores (Handlers):** Conjunto de funciones que gestionan los eventos internos y externos, (los externos son los producidos por la llegada de una

notificación al sistema). Cada función sólo se ejecuta cuando llega una notificación desde el puerto donde está asociado el manejador. Los manejadores se asocian a los puertos de tipo notificación.

- **Notificaciones (Notifications):** Mensajes que son recibidos desde los distintos puertos y que se producen cuando se ha generado un evento en un servicio. Por ejemplo, Cuando un objeto golpea un bumber, se produce la recepción de un mensaje que ha sido generado por un evento que indica que uno de los bumpers ha sido golpeado. (Este mensaje desencadenará normalmente la ejecución de un manejador).

O de una manera más resumida, podemos decir que un servicio está compuesto de cuatro componentes, que agrupan a todos estos ahora descritos, y serían:

- **Contrato**
- **Estado interno**
- **Comportamiento**
- **Contexto de ejecución**

2.2.1.1 Contratos

Un identificador de contrato se usa para identificar los diferentes servicios. Cuando nosotros creamos uno, automáticamente se crea un identificador. En el aspecto de la programación, cada servicio irá dentro de una clase que será conocida como ***Contract*** y dentro de ella habrá un campo llamado ***Identifier*** que será el que haga referencia a la URI que será nuestro identificador de contrato. La URI será representada de la siguiente manera:

http://somehost.domain/path/year/month/servicename.html

Esta URI no existe en internet aunque por el nombre pueda parecerlo. Lo importante es que tiene que ser una referencia única por cada servicio.

IMPORTANTE

Un mismo servicio puede tener varios suplentes (contratos). Los contratos alternativos permiten compartir datos sin hacer peticiones explícitas entre ellos. Precaución: se usa normalmente para leer información sobre el estado de otros servicios, pero no para modificarlo. En vez de eso, se usan peticiones update o Request al otro servicio y se le deja hacer la modificación.

2.2.1.2 Estado

El estado interno de un servicio es una clase que contiene las propiedades que serán importantes para realizar operaciones con ellos. Algunas de sus propiedades las vamos a considerar constantes para la duración de su ejecución o también pueden representar cosas que ocurren en el mundo real como por ejemplo, los valores obtenidos por medio de un sensor.

El estado podemos almacenarlo como estado guardado o en un archivo XML que podemos cargarlo las veces que queramos y donde recuerda los parámetros.

2.2.1.3 Comportamiento

El comportamiento describe lo que realmente hace el servicio. Incluye algoritmos que un servicio usa para lograr su propósito. Posee operaciones que no tienen por qué enviar una respuesta. En el caso de que se crea que una operación pueda fallar, tenemos que especificar un puerto de respuesta y crear una relación de causalidad. Utilizamos el servicio de los handler para procesar las operaciones (la mayorías se controlan por medio de interlieve).

2.2.1.4 Contexto de ejecución

Es importante saber que el contexto de un servicio va a incluir a todos sus socios (partners). La asociación que va a haber entre el servicio actual y otro servicio, se va a realizar usando un manifiesto, que es un fichero XML, como por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<Manifest
xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
      >
  <CreateServiceList>
    <!--Start BasicStamp2 Brick -->
    <ServiceRecordType>
dssp:Contract>http://schemas.microsoft.com/robotics/2007/06/basic
stamp2.html
      </dssp:Contract>
      <dssp:PartnerList>
        <!--Initial BasicStamp2 config file -->
        <dssp:Partner>
          <dssp:Service>Parallax.BoeBot.Config.xml</dssp:Service>
          <dssp:Name>dssp:StateService</dssp:Name>
        </dssp:Partner>
      </dssp:PartnerList>
    </ServiceRecordType>
```

```
    <!--Start the BoeBot drive service-->
<ServiceRecordType>
<dssp:Contract>http://schemas.microsoft.com/robotics/2007/06/bsd
rive.html
    </dssp:Contract>
    <dssp:PartnerList>
     <!--Initial Drive Configuration File -->
     <dssp:Partner>
      <dssp:Service>Parallax.BoeBot.Drive.Config.xml</dssp:Service>
      <dssp:Name>dssp:StateService</dssp:Name>
     </dssp:Partner>
    </dssp:PartnerList>
   </ServiceRecordType>
   <!-- Dashboard -->
   <ServiceRecordType>
<dssp:Contract>http://schemas.microsoft.com/robotics/2006/10/das
hboard.html
</dssp:Contract>
   </ServiceRecordType>
  </CreateServiceList>
</Manifest>
```

Esta estructura va a representar el esqueleto que deben seguir todos los ficheros XML en los que queramos definir una lista de servicios asociados a nuestro proyecto. Dentro de dicha lista, haremos la llamada al servicio que queramos incluir, poniendo su fichero XML junto con su nombre y su identificador de contrato.

2.2.2 DSSP (Protocolo DSS)

DSSP define una aplicación como una composición de servicios que pueden ser aprovechados para lograr una tarea deseada a través de la orquestación. Los servicios son entidades que podemos crear, manipular, controlar y destruir varias veces durante el ciclo de vida de una aplicación. Para ello usaremos las operaciones definidas por el DSSP. Algunas de ellos son:

- Create: Crea un nuevo servicio.
- Delete: Borra parte de un estado de un servicio.
- Drop: Termina un servicio.
- Get: recupera una copia del estado del servicio.
- Insert: Añade información al estado del servicio.
- Lookup: Recupera información sobre el servicio y su contexto
- Query: Similar a Get, solo se diferencia en que debemos de pasarle parámetros.

- Replace: Reemplaza el estado del servicio.
- Subscribe: Solicita notificaciones para todos los cambios del estado.
- Submit: Permite cálculos que no modifiquen el estado.
- Update: Recupera la información de un servicio y su contexto.
- Upsert: Actualiza el estado existente o añade un nuevo estado para describirlo.

Como podemos ver, se Incluyen muchas clases de operaciones, pero la única obligatoria de acuerdo al protocolo DSS (DSSP) es Lookup. Pero un servicio solo con esta operación sería inservible, de modo que debemos incluirla junto con Drop y Get.

2.2.3 Creación de un proyecto

Una vez realizada instalación y tras la explicación realizada en capítulos anteriores, es hora de crear un nuevo proyecto. Para ello, deberemos empezar abriendo Microsoft Visual Studio, ya que vamos a tener una serie de plantillas que nos van a facilitar el trabajo a la hora de crear un nuevo servicio para nuestro proyecto.

Tenemos que seguir los siguientes pasos:

1. Una vez abierto Microsoft Visual Studio, seleccionamos Archivo -> Nuevo -> Proyecto
2. En el marco derecho, marcamos Plantillas instaladas->Visual C# -> Microsoft Robotics y aparecerá en el marco central, la plantilla que vamos a utilizar, en nuestro caso DSS Service (4.0)
3. En la parte inferior, le daremos nombre a nuestro proyecto, seleccionaremos su ubicación donde queramos que se encuentre, y el nombre a la solución y pulsamos aceptar. (Figura 2.2.3-1)

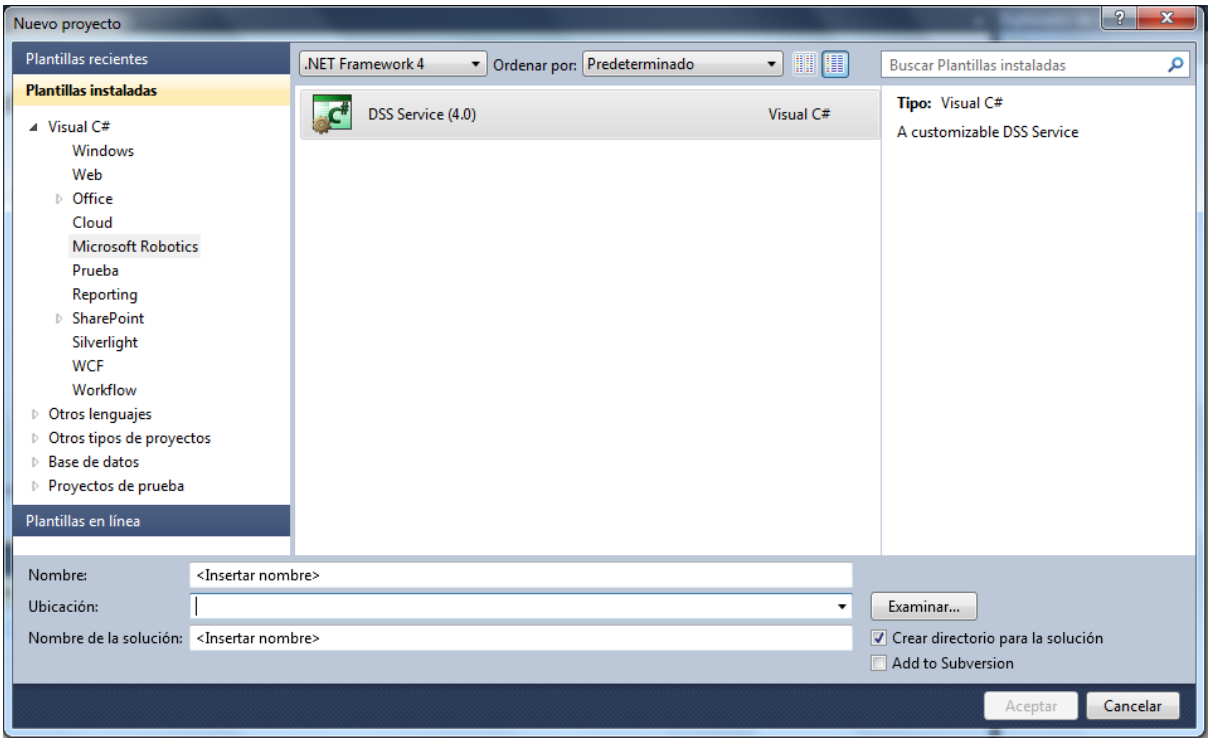


Figura 2.2.3-1 Crear un proyecto

4. En la siguiente ventana que nos aparezca, indicaremos cual será la URI y aceptamos. (Figura 2.2.3-2)

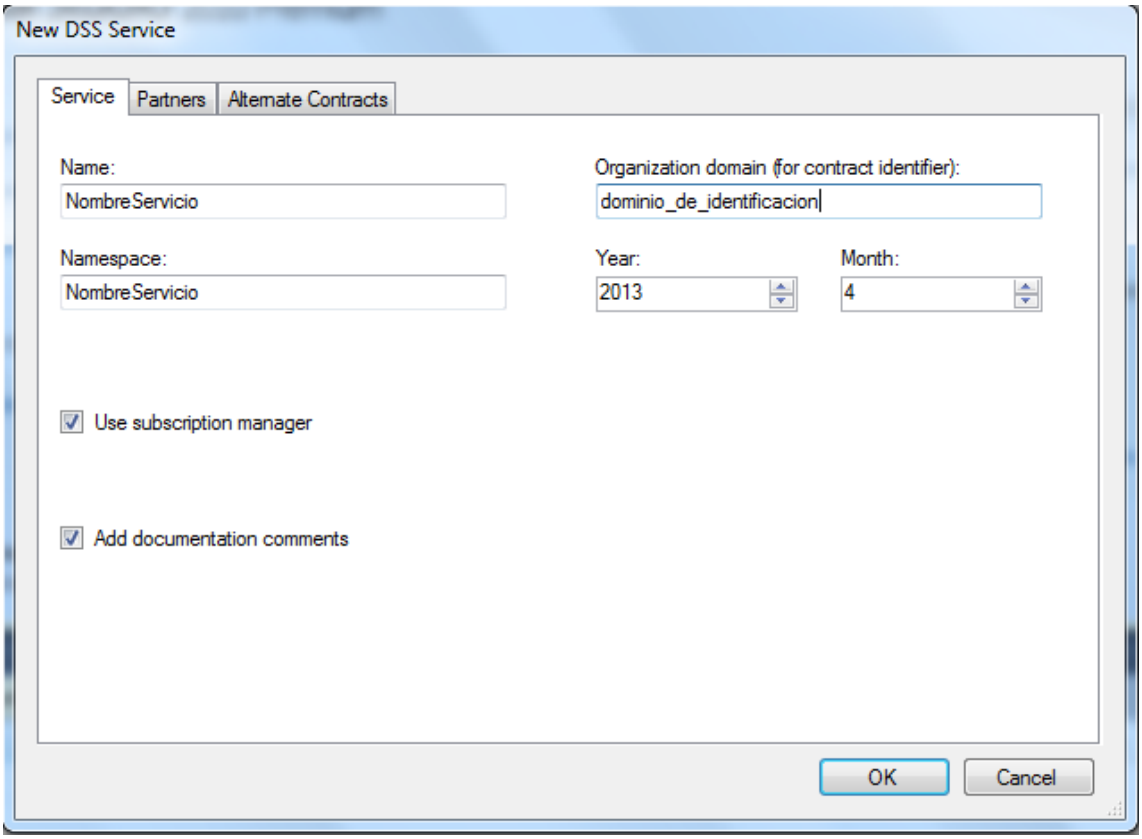


Figura 2.2.3-1 URI

Una vez finalizada la creación del proyecto, podemos observar a la derecha la lista de soluciones en “Explorador de soluciones” y cuyas propiedades tienen que ser:

- Aplicación: es la biblioteca de clases DLL. El nombre del ensamblado y el identificador del contrato deben coincidir.
- Depurar: si nuestra versión es de 64bits, debemos cambiar el programa externo de inicio a DssHost32
- También deberemos especificar la ruta correcta hacia el directorio de trabajo donde esta nuestro manifiesto.

  /p:50000 /t:50001

  /m:"../Nuestra_Ruta_Hacia_El_Directorio/NombreServicio.manifest.xml"

La lista de soluciones es la siguiente:

- **AssembyInfo.cs:** Va a ser quien contenga la información sobre el ensamblado DLL, y es ajeno al control del usuario.
- **NombreServicio.cs**: Es el archivo principal sobre el que vamos a trabajar.

A continuación vamos a mostrar un código genérico que servirá para que podamos ver cómo son las clases .cs en un ámbito global.

```
namespace NombreServicio
{
    [Contract(Contract.Identifier)]
    [DisplayName("NombreServicio")]
    [Description("NombreServicio service (no description
provided)")]
    class NombreServicioService : DsspServiceBase
    {
        /// <summary>
        /// Service state
        /// </summary>
        [ServiceState]
        NombreServicioState _state = new NombreServicioState();

        /// <summary>
        /// Main service port
        /// </summary>
        [ServicePort("/NombreServicio", AllowMultipleInstances =
true)]
        NombreServicioOperations _mainPort = new
NombreServicioOperations();

        [SubscriptionManagerPartner]
        submgr.SubscriptionManagerPort _submgrPort = new
submgr.SubscriptionManagerPort();

        /// <summary>
        /// Service constructor
        /// </summary>
        public NombreServicioService(DsspServiceCreationPort
creationPort)
```

```
            : base(creationPort)
        {
        }

        /// <summary>
        /// Service start
        /// </summary>
        protected override void Start()
        {

            //
            // Add service specific initialization here
            //

            base.Start();
        }

        /// <summary>
        /// Handles Subscribe messages
        /// </summary>
        /// <param name="subscribe">the subscribe request</param>
        [ServiceHandler]
        public void SubscribeHandler(Subscribe subscribe)
        {
            SubscribeHelper(_submgrPort, subscribe.Body,
subscribe.ResponsePort);
        }
    }
}
```

- Algunas partes claves como la declaración del estado, el puerto principal y el constructor del servicio vienen detalladas a continuación. Nótese que indicamos que no se permite crear por duplicado la instancia del servicio con "AllowMultipleInstances = false".
- Display y descripción son solo para la documentación.
- Debemos tener una instancia a nuestro servicio (aunque este vacío) y un puerto principal de operaciones, new State , mainPort ServiceOperations -→ ambos con el nombre que queramos. Este mainPort es para mandarnos mensajes a nosotros mismos.
- Constructor: siempre está vacío.
- Start: es el método que se ejecuta al iniciar el servicio. Incluye el base.Start. Al ejecutar el manifiesto, se pasa a DssHost
- Handler: lo utilizamos para el Get.

  - **NombreServicio.Manifest.xml:** es el manifiesto usado por el programa para inicializar y cargar el servicio.

```
<?xml version="1.0" ?>
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"

xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    >
```

```
    <CreateServiceList>
        <ServiceRecordType>

<dssp:Contract>http://dominio_de_identificacion/2013/04/nombreser
vicio.html</dssp:Contract>
        </ServiceRecordType>
    </CreateServiceList>
</Manifest>
```

- **NombreServicioTypes.cs**: Aquí estará definida la composición del estado del servicio (todas las clases y tipos de datos).

```
        /// <summary>
        /// NombreServicio contract class
        /// </summary>
        public sealed class Contract
        {
            /// <summary>
            /// DSS contract identifer for NombreServicio
            /// </summary>
            [DataMember]
            public const string Identifier =
"http://dominio_de_identificacion/2013/04/nombreservicio.
html";
        }

/// <summary>
        /// NombreServicio state
        /// </summary>
        [DataContract]
        public class NombreServicioState
        {
        }

        /// <summary>
        /// NombreServicio main operations port
        /// </summary>
        [ServicePort]
        public class NombreServicioOperations :
PortSet<DsspDefaultLookup, DsspDefaultDrop, Get,
Subscribe>
        {
        }

        /// <summary>
        /// NombreServicio get operation
        /// </summary>
        public class Get : Get<GetRequestType,
PortSet<NombreServicioState, Fault>>
        {
            /// <summary>
            /// Creates a new instance of Get
            /// </summary>
            public Get()
            {
            }

            /// <summary>
```

```
        /// Creates a new instance of Get
        /// </summary>
        /// <param name="body">the request message
body</param>
        public Get(GetRequestType body)
            : base(body)
        {
        }

        /// <summary>
        /// Creates a new instance of Get
        /// </summary>
        /// <param name="body">the request message
body</param>
        /// <param name="responsePort">the response port
for the request</param>
        public Get(GetRequestType body,
PortSet<NombreServicioState, Fault> responsePort)
            : base(body, responsePort)
        {
        }
    }
```

- Clase Contract: El identificador del contract es único y se crea por defecto.

- Estado del servicio: Todos los servicios tienen una clase estado, pero al principio se crea vacía. Tenemos que implementarlo de tal manera, que el estado se pueda guardar y, cuando volvamos a usar el servicio, tengamos el estado tal y como lo guardamos. Hay que añadir campos al estado para que pueda ser accedido y para el envío de mensajes.

  Se hace un [DataContract] por cada clase que queramos crear. (Las que queremos que salgan en el proxy).

- Puerto Principal de operaciones: [ServicePort]

  Los servicios tienen un atributo ServicePort, en el que se define la clase de las opraciones: NombreServicioOperations: PortSet<*Aquí se declaran las operaciones que nuestro servicio va a poseer*>

  Por defecto, nuestro servicio llevará las operaciones: DefauktLookUp, DsspDefaultDrop,Get. Pero podemos añadir las que queramos y luego implementarlas.

  - DsspDefaultLookup: es una manera de anunciar su defunción de servicio, es lo que usa el runtime de DSS para saber las operaciones del servicio.
  - DsspDefaultDrop: es la operación de eliminar servicio.
  - Get, esta operación se puede usar para muchas cosas pero en principio es para obtener el estado actual del servicio, ósea obtener el estado.
  - Transferir y Conectar son operaciones personalizadas.

- En la clase Get se definen las operaciones de solicitud.

Public class Get: Get <GetRequestype,PortSet< NombreServicioState
,Fault>{}

GetRequesttype: Envio de solicitud

ServiceState/Fault: se devolverá el estado del servicio en cuestión o error.

3. Implementación de diferentes tipos de sistemas

Para facilitar la simulación de sistemas, hemos creado una plantilla con la cual no hará falta ser un experto programador para implementar un sistema con MRDS. Con esta plantilla, sólo hay que introducir las ecuaciones del sistema en el módulo correspondiente para su correcta simulación.

Nuestra plantilla, tendrá más o menos esta apariencia:

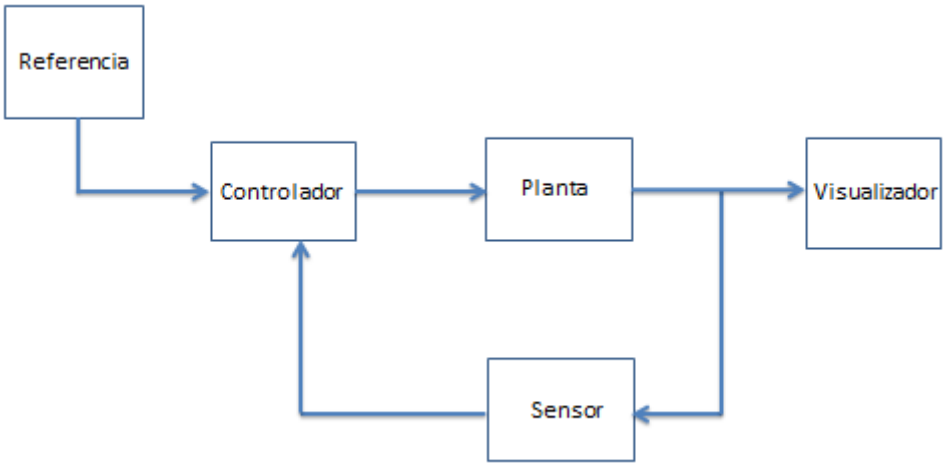


Figura 3-1 Plantilla del sistema de módulos

Cada uno de estos módulos, se ha construido con la siguiente funcionalidad:

- **Referencia:** Este módulo será la fuente de datos que introduciremos en el sistema, bien para que sean modificadas, o para usarlas para hacer nuevos cálculos. Estos datos serán introducidos manualmente durante la ejecución del proyecto, o indicándolo explícitamente en el código.
- **Controlador:** Si queremos que los cálculos de nuestro sistema estén sujetos a algunas condiciones, se deberán introducir éstas en este módulo. A él llegarán los datos que el sensor tome de la planta, y los datos de entrada al sistema, y con ellos se calculará el valor que habrá de recibir la planta.
- **Planta:** En este módulo deberán introducirse las ecuaciones del sistema a implementar. La planta tomará los datos de entrada que le lleguen del controlador, y realizará los cálculos necesarios.
- **Sensor:** Este módulo medirá la salida de planta, para pasársela como entrada al controlador.
- **Visualizador:** Este módulo servirá para que el programador pueda ver las salidas del sistema sin necesidad de usar la consola. Se pueden usar por

ejemplo gráficas para visualizar los datos de salida en función de los datos de entrada.

Para implementar cualquier sistema con nuestra plantilla, simplemente habrá que sustituir las ecuaciones y las condiciones que tenemos declaradas en nuestros ejemplos, y declarar las variables de entrada y salida que necesitemos en cada servicio, como variables de estado de cada módulo.

Para la implementación de los módulos generales, y el paso de mensajes entre ellos, hemos utilizado operaciones de tipo Update[10], que hacen que cuando un servicio actualice su estado, el cambio se envíe a todos los servicios que están suscritos a él. Cada uno de estos suscriptores tendrá una operación handler, en la que se defininirá el comportamiento que tendrá el servicio cada vez que reciba un cambio en sus servicios socios[11].

Para demostrar el funcionamiento de nuestra plantilla, hemos planteado una serie ejemplos de distintos tipos de sistemas (discretos, continuos y por eventos), para mostrar cómo se aplicarían a nuestro proyecto.

A continuación explicamos en detalle cada uno de estos ejemplos.

3.1 Sistema discreto

Un sistema discreto es aquel en el cual, sus señales son consideradas, o existen sólo a intervalos discretos de tiempo. Suelen ser resultado de un muestreo de señales continuas. En este caso no usaremos un sistema muestreado sino que presentamos un sistema discreto muy simple que sirve de introducción a los módulos diseñados.

Como ejemplo de Sistema Discreto, hemos diseñado un proyecto, que calcula la media de una serie de valores aleatorios, y que controla que el valor de esta media no supere un valor máximo fijado.

Nuestro sistema, siguiendo la plantilla anterior; está dividido en cuatro módulos (servicios MRDS): Referencia, Controlador, Planta y Sensor.

Las funciones de cada uno de ellos son las siguientes:

- **Referencia:** Encargada de proporcionarle al sistema, los valores para los que calcularemos la media.
- **Controlador:** Es el módulo encargado de que la media no supere un valor máximo que fijaremos nosotros.
- **Planta:** Es el encargado de ir calculando la media con los valores que vaya recibiendo y mostrar ésta por consola.
- **Sensor:** Encargado de ponderar la media (la multiplica por 0,9).

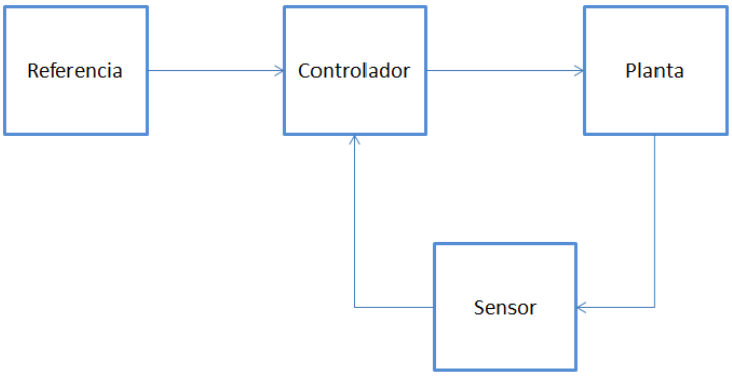


Figura 3.1 -1 Esquema de módulos del sistema discreto

Por simplicidad, en este primer ejemplo no utilizaremos el módulo visualizador sino que se mostrarán por consola los valores de la media que se vayan calculando, así como los valores que vamos introduciendo, y el número de cálculos que se han realizado.

A continuación se explican más detalladamente cada uno de los módulos:

3.1.1 Módulo Referencia

Este módulo, posee una operación (GetReference), que genera números aleatorios comprendidos entre [1-10]. Cuando el controlador lo solicite, la referencia generará un nuevo número que será el nuevo valor que usaremos para recalcular la media.

Como variables de estado tenemos

- Numero: Donde almacenaremos el siguiente número para el que se va a calcular la media. Esta variable será accedida por el controlador cuando tenga que generar un nuevo valor para el cálculo
- Máximo: A través de una interfaz que se mostrará al lanzar el servicio referencia, el usuario podrá escoger un valor máximo que la media no va a sobrepasar. El controlador usará este valor para sus cálculos.

El módulo referencia tendrá sólo un socio (Controlador), que accederá a sus variables de estado para obtener la información que necesite.

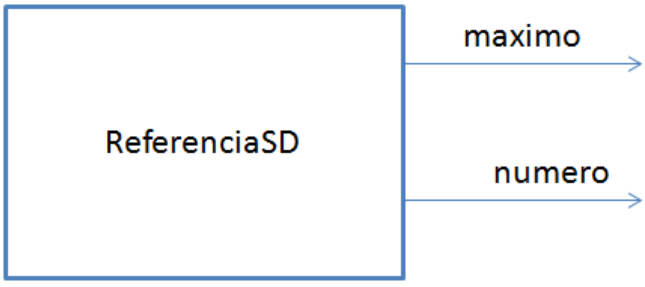


Figura 3.1.1 -1 Comportamiento de ReferenciaSD de Sistema Discreto

3.1.2 Módulo Controlador

El módulo controlador se encarga de que el valor de la media no sobrepase un valor fijo (en nuestro ejemplo 8), pasándole a la planta el valor adecuado para calcular la nueva media.

El controlador, tendrá como entradas la salida del sensor, y las salidas de referencia, por tanto; tiene a éstos declarados como socios. Para calcular el valor nuevo adecuado, y pasar éste a la planta, el controlador comparará el valor que le venga de Sensor (que será el valor de la media actual multiplicado por 0,9), con el valor máximo que tome de referencia. En caso de que la media supere el máximo, el controlador fijará como siguiente número para recalcular la media un 0 (así ésta bajará considerablemente), y en caso contrario, pedirá a referencia que genere un nuevo número aleatorio, y este será el nuevo valor.

Para realizar todas estas operaciones, en el controlador hemos declarado las siguientes variables de estado: máximo, mactual, vnuevo y vref.

- maximo: Se almacenará aquí el valor máximo que queramos para la media, éste se cogerá de Referencia cuando el controlador lo necesite.
- mactual : En esta variable guardaremos el valor de la media actual que vendrá de Sensor.
- vref: En esta variable almacenaremos el valor aleatorio que nos dé referencia cuando se lo pidamos
- vnuevo: Cuando calculemos el nuevo valor para recalcular la media, éste será almacenado aquí y será enviado a Planta.

Figura 3.1.2 -1 Comportamiento de ControladorSD de Sistema Discreto

3.1.3 Módulo Planta

En el módulo Planta, realizaremos el cálculo de la media. Cada vez que le llegue un nuevo valor de Controlador, la planta realizará la siguiente operación:

$$media(k+1) = \frac{1}{k+1} * v(k) + \frac{k}{k+1} * media(k)$$

Donde media(k), será el valor de la media anterior que tengamos almacenado, media (k+1) será el valor nuevo de la media recalculado para el valor v(k), que será el que nos venga de Controlador y por último, k será una variable interna que llevará la cuenta de las iteraciones. Para cada k, la media tomará un valor, y a medida que k se vaya incrementando, menos afectarán los valores que nos lleguen para el cálculo.

Como variables de estado tendremos:

- pmedia: Variable en la que almacenaremos la media actual. Será el valor que se pasará a sensor, y el que se utilizará en cada iteración para calcular la media nueva.
- pvalor: Variable de entrada en la que se almacenará el valor que le venga de controlador.
- k : Variable interna que llevará la cuenta del número de iteraciones, y se incrementará en 1 cada vez que hagamos un nuevo cálculo de la media.

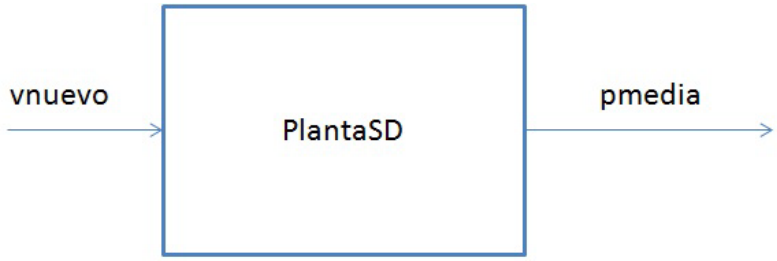


Figura 3.1.3 -1 Comportamiento de PlantaSD de Sistema Discreto

Cada vez que la Planta calcule un nuevo valor de la media, podremos ver en la consola, el valor de la nueva media, los números que hemos ido introduciendo para calcularla, y el número de iteración k:

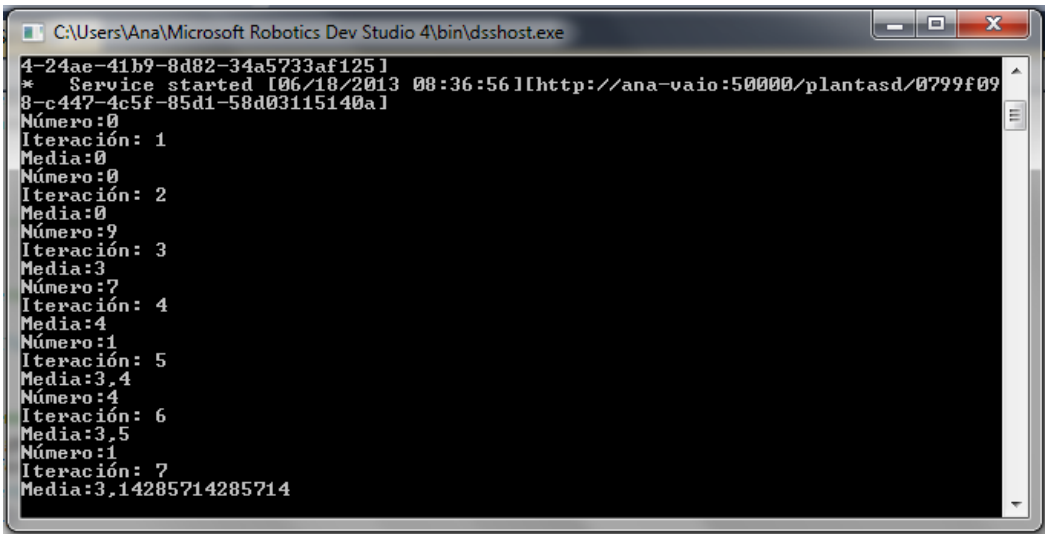


Figura 3.1.3 -2 Salida por consola de Sistema Discreto

3.1.4 Módulo Sensor:

Este módulo se encarga de ponderar la media. Así, cada vez que reciba una notificación de planta con el nuevo valor de la media, multiplicará este valor por 0,9 y enviará la media ponderada al controlador.

Tendremos dos variables de estado:

- mediaplanta : Variable de entrada en la que se almacenará el valor actual de la media en la planta
- mediapond: Cuando se pondere la media el nuevo resultado se almacenará aquí, y será la variable que pasará al sensor para que éste realice sus cálculos.

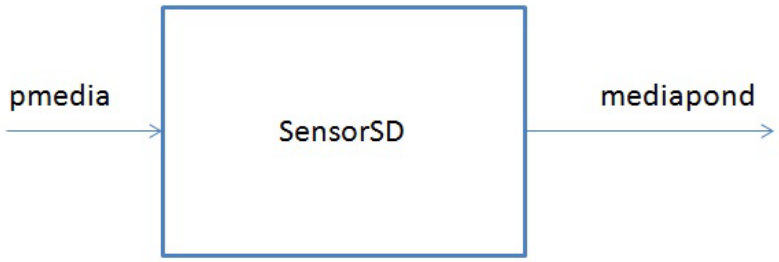


Figura 3.1.4 -1 Comportamiento de SensorSD de Sistema Discreto

Para arrancar este sistema, ponemos en marcha el controlador, que arrancará a sensor y referencia, y sensor a su vez arrancará a Planta. Así tendremos todos los servicios en marcha. Para inicializar Controlador, actualizamos su estado (la variable *numero*), con un cero. Así este valor pasará a la planta, que calculará la media con la fórmula antes dada, y enviará la correspondiente notificación al sensor. Este multiplicará el valor de la media por 0,9, y mandará una notificación al controlador para informarle de que ha actualizado su estado. En las siguientes iteraciones, cuando el Controlador reciba el

valor del sensor, comparará éste con el valor MAX que le hayamos indicado (en este caso 8). Si el valor es menor que Max, entonces el controlador solicitará la operación getReference, para que ésta le envíe un número aleatorio (que estará comprendido entre 1 y10). Si el valor que llega del sensor es superior a MAX, el controlador directamente actualizará su estado con un 0 para bajar la media. Este proceso se repetirá hasta que paremos el servicio.

3.2 Sistema Continuo

Los sistemas continuos son aquellos en que las variables son continuas a lo largo del tiempo.

Como ejemplo de Sistema Continuo, hemos creado un proyecto que se encargue de llevar el control de la temperatura en una habitación, de tal manera que ésta se mantenga en todo momento a la temperatura deseada. Para ello llevaremos un seguimiento continuo de la temperatura a la que se encuentra la habitación, y le suministraremos calor en caso de que dicha temperatura se encuentre por debajo de la deseada.

El sistema está dividido en seis módulos: TiempoSC, ControladorSC, ReferenciaSC, PlantaSC, SensorSC y VisorSC. El funcionamiento básico de cada uno es el siguiente:

- **TiempoSC:** Este, será el servicio gestor del tiempo. Desde él, se irán poniendo en marcha todos los demás cuando les corresponda. En él, llevaremos un control del tiempo de tal manera que pueda lanzar, en el instante que le toca, al servicio correspondiente.
- **ControladorSC:** Este módulo se encarga de comprobar si la temperatura actual de la habitación se encuentra por debajo de la temperatura deseada en ese momento, y calculará el valor del calor que ha de suministrarse en tal caso.
- **ReferenciaSC:** La referencia se encarga de proporcionar al controlador la temperatura deseada en un momento dado.
- **PlantaSC:** Es el módulo encargado de calcular la nueva temperatura añadiendo el calor calculado en el controlador.
- **SensorSC:** Módulo encargado de medir la temperatura de la habitación y enviársela al controlador.
- **VisorSC:** Muestra una gráfica con la evolución de la temperatura.

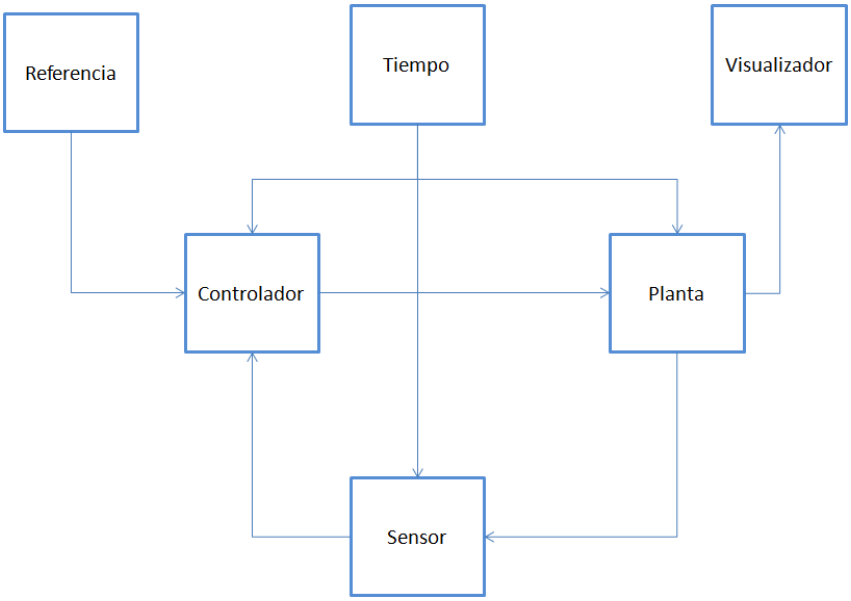


Figura 3.2 Esquema de módulos del sistema continuo

3.2.1 Módulo Referencia:

Este módulo será el encargado de introducir en el sistema, la temperatura deseada de cada momento, o el calor que se desea suministrar a la habitación.

Para ello, al arrancar el servicio Referencia, al usuario se le mostrará una interfaz en la que podrá elegir entre dos modos: Modo Automático y Modo Manual.

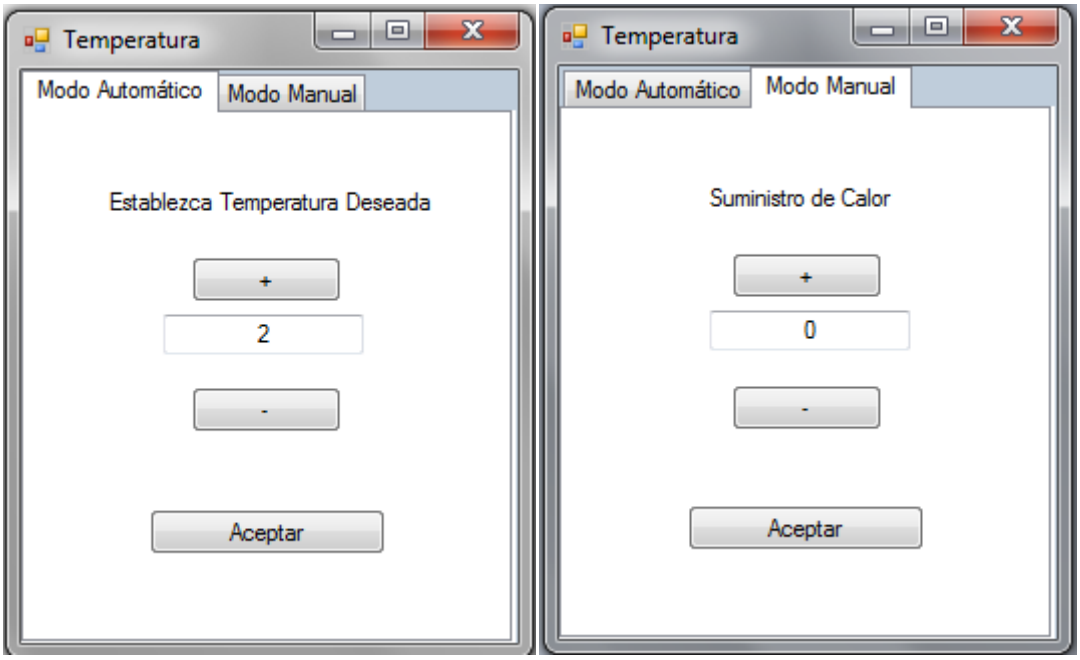


Figura 3.2.1 -1 Interfaz de ReferenciaSC del sistema conitnuo.

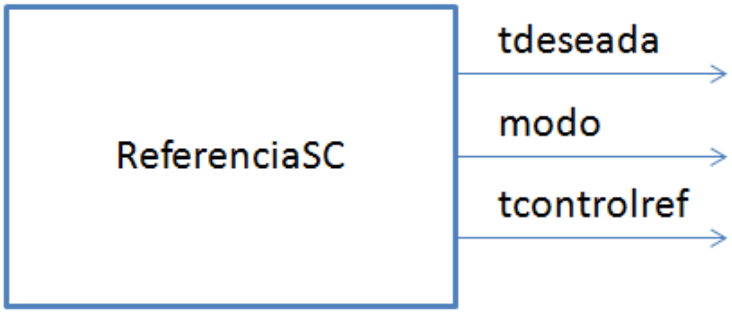
En caso de escoger Modo Automático, el usuario podrá escoger la temperatura a la que quiere que esté la habitación, aumentando o disminuyendo los grados (en un intervalo de [2-30]), y pulsando el botón Aceptar. El valor de la temperatura escogida

será almacenado en una variable de estado para poder ser accedida desde el módulo controlador.

En el caso del Modo Manual, el usuario deberá elegir el calor que quiere suministrar a la habitación. Como en el caso anterior, deberá introducir el calor deseado, y pulsar el botón de aceptar para que esto se guarde en otra variable de estado.

Para gestionar los datos de este módulo, contaremos con tres variables de estado: modo, tdeseada, tcontrol.

- modo: Variable booleana que almacenará el modo con el que estamos operando (Si es 1 será modo automático, y si es 0 será modo manual)
- tdeseada: Una vez confirmemos la elección de la temperatura en el modo automático, ésta se almacenará aquí para poder ser accedida por el controlador cuando éste lo desee.
- tcontrol: Variable donde se almacenará el calor que deseamos suministrar, en caso de que nos encontremos en modo manual.



Figuro 3.2.1 -2 Comportamiento de ReferenciaSC de Sistema Continuo.

3.2.2 Módulo Tiempo

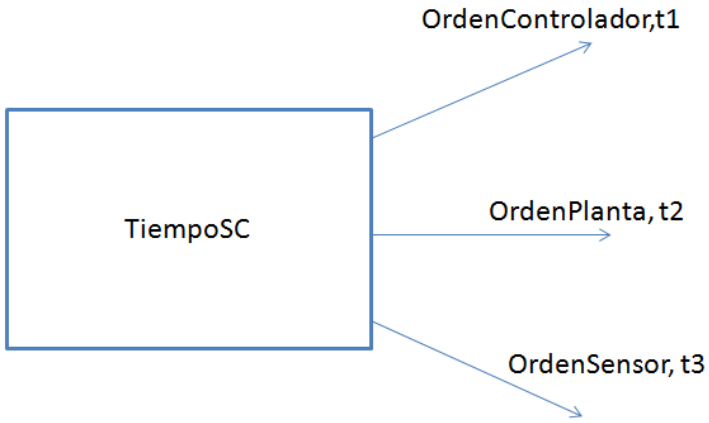
En este ejemplo, el módulo del tiempo será el módulo principal. La funcionalidad de este módulo es evitar que el resto de servicios, estén continuamente realizando cálculos y para esto lo que hace es lanzar cada uno de los servicios cada cierto tiempo.

Su funcionamiento es el siguiente: El servicio del tiempo tiene como socios a los servicios Controlador, Planta y Sensor, de tal manera que puede mandarles realizar operaciones que se encuentren definidas en cada uno de ellos. Para hacer esto, se han definido tres contadores (timers) , que se encargan de lanzar eventos cada x segundos. Cada uno de estos eventos llevará un código asociado que indicará al módulo tiempo lo que ha de hacer. Así cuando se lance el evento correspondiente a alguno de los servicios, lo que se hará será enviar una petición “Update”, al puerto del servicio que le

corresponda. De esta manera, el servicio correspondiente realizará sus cálculos con los valores que tenga almacenados en sus variables de estado, actualizará su estado con los nuevos valores.

Con este método evitamos realizar cálculos inútiles (sin él realizaremos varias veces los mismos cálculos sobre los mismos valores). Cada vez que uno de los módulos realice algún cálculo, y envíe el resultado a sus servicios asociados, este valor quedará almacenado en dichos servicios, pero no se operará con él hasta que el servicio Tiempo no se lo mande.

En este servicio tendremos tres variables de estado: t1, t2, t3, que almacenarán la cantidad de segundos que van pasando en los respectivos eventos e1 , e2 y e3.



Figuro 3.2.2-1 Comportamiento de TiempoSC de Sistema Continuo.

3.2.3 Módulo Controlador

En el módulo controlador, se calculará el calor que hay que suministrarle a la habitación para conseguir llegar a la temperatura deseada.

Para realizar estos cálculos, el controlador tendrá como socios (“Partners”) a los servicios Referencia y Sensor. Del primero recibirá la temperatura deseada cada vez que se modifique este valor, y el calor que debemos suministrar, en caso de encontrarnos en modo manual, para saber el modo en que nos encontramos, también recibirá el valor de la variable “modo” de la referencia. De Sensor recibirá el valor de la temperatura actual que hay en la habitación. Cada vez que alguno de estos dos módulos actualice su estado, una notificación llegará al controlador, que almacenará los valores que le lleguen pero no hará nada con ellos. Sólo en el momento en que el módulo del tiempo mande al controlador actualizarse, éste calculará el calor que ha de ser suministrado, con los últimos valores que tenga almacenados de sus servicios socios.

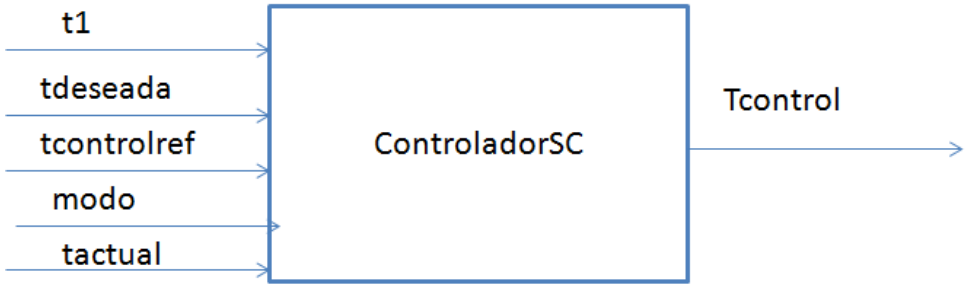
Si nos encontramos en modo automático, cada vez que Controlador (que implementa un PID) reciba la orden de actualizarse, llevará a cabo las siguientes operaciones para calcular la temperatura de control que ha de ser enviada a la planta.

error = Tdeseada – Treal;
*control = k_p * error + integral + derivativo;*
*integral = integral + k_i *error * Tmuestreo;*
derivativo = (k_d/Tmuestreo)(error-error_previo)*
error_previo = error;

Todas estas variables, serán declaradas como variables de estado, y la variable control, será la que se le pase a planta para que haga el nuevo cálculo de la temperatura de la habitación.

En caso de estar en modo manual, la temperatura que enviaremos a la planta será directamente la temperatura de control que el controlador ha recibido de referencia, y que tendrá almacenado en otra variable de estado llamada tcontrolref.

Todas estas variables, serán declaradas como variables de estado, y la variable control, será la que se le pase a planta para que haga el nuevo cálculo de la temperatura de la habitación.



Figuro 3.2.3 -1 Comportamiento de ControladorSC de Sistema Continuo

3.2.4 Módulo Planta

En este módulo se hará el cálculo de la temperatura actual a partir de la siguiente ecuación:

$$Temperatura = Tatm + T_0 * e^{-t/\tau} + \left(1 - e^{-\frac{t}{\tau}}\right) * T_c$$

Donde Tatm es la temperatura ambiente del momento actual y Tc es el calor que ha de suministrarse, que vendrá de Controlador.

Como variables de estado en este servicio, tendremos:

- controlp : Donde se almacenará el valor de control(Tc en la ecuación anterior) cada vez que el controlador haga una actualización de su estado.
- treal : Aquí se almacenará el nuevo valor de la temperatura calculado, para enviárselo al Sensor, y disponer de él en el próximo cálculo.

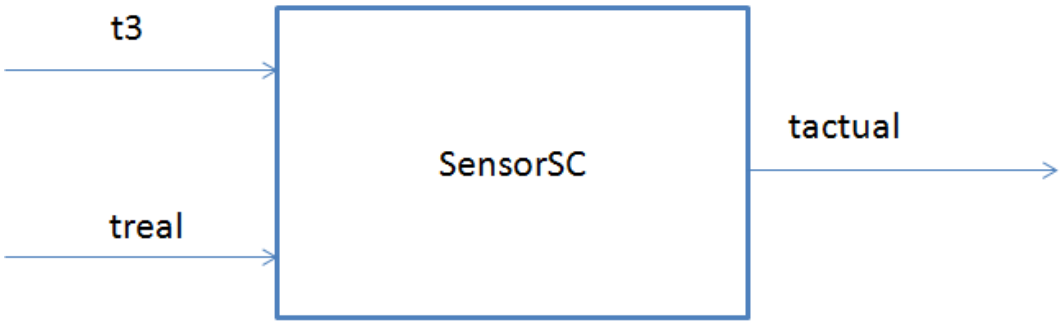
Figuro 3.2.4 -1 Comportamiento de PlantaSC de Sistema Continuo

Del mismo modo que el controlador, la planta almacenará los valores que le vengan de su servicio asociado (Controlador), cada vez que éste actualice su estado, pero no realizará los cálculos para la temperatura nueva hasta que el módulo tiempo no se lo ordene.

3.2.5 Módulo Sensor

El módulo Sensor se encargará de medir la temperatura que hay en Planta, y enviársela al Controlador cuando reciba la orden del Módulo tiempo.

Tendrá una variable de estado (temperatura), que será donde se almacene la temperatura de planta cada vez que ésta le envíe una notificación, y esta misma variable será la que se envíe a Controlador cuando el sensor actualice su estado.



Figuro 3.2.5 -1 Comportamiento de SensorSC de Sistema continuo

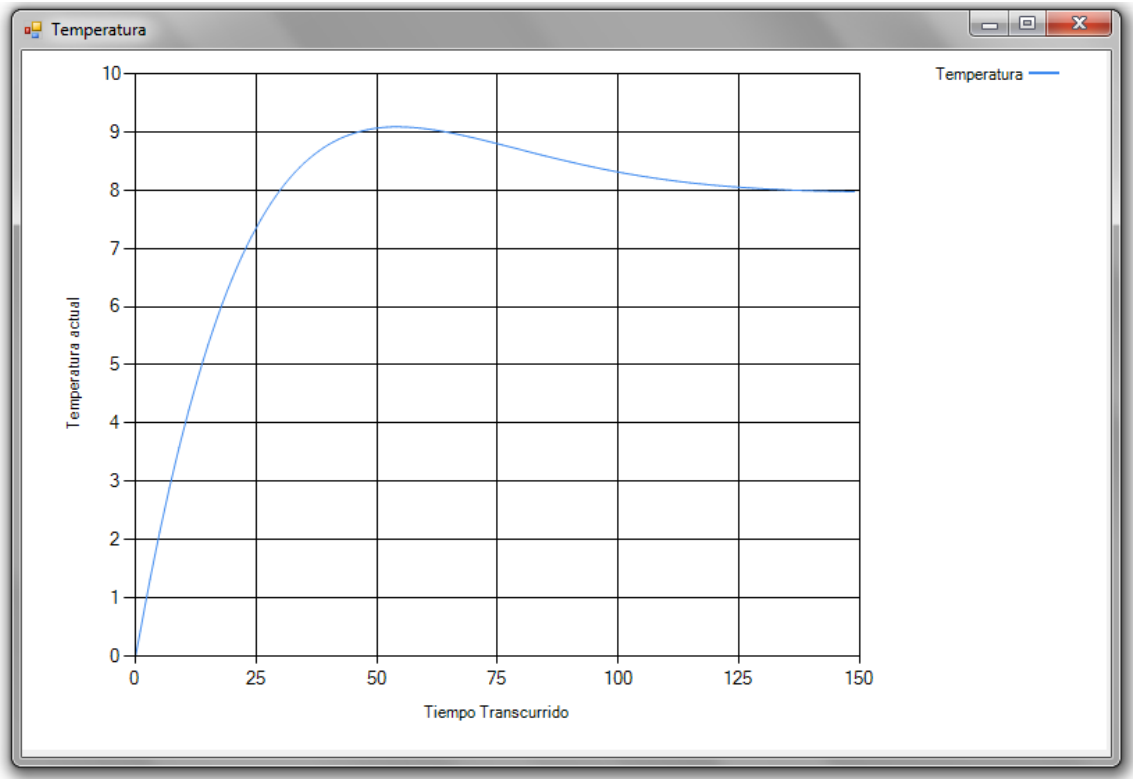
3.2.6 Módulo Visualizador

Este servicio mostrará una gráfica[12], con la evolución de la temperatura en el tiempo.

Cada punto (X,Y) se corresponderá con la temperatura actual (x) en el instante actual(y). Para obtener estos puntos, el visualizador recibirá información del estado de planta, que le dirá la temperatura correspondiente a un instante t que será el instante en el que el módulo gestor de tiempo le ordenó que actualizara la temperatura.

Ejemplo para una temperatura deseada de ocho grados, con una temperatura inicial de cero grados.

Podemos ver como con el tiempo, la temperatura irá aumentando hasta llegar a sobrepasar los ocho grados, pero después se estabilizará.



Figuro 3.2.6 -1 Salida del Sistema continuo

3.3 Sistemas basados en eventos: DEVS (Discrete Events System Specification)

Dentro de DEVS[9] podemos encontrarnos con dos modelos de sistemas basados en eventos, atómicos y acoplados. Nosotros nos centraremos en el modelo atómico. El modelo atómico es el que especifica y experimenta con el comportamiento dinámico. Sin embargo si juntamos estos modelos pueden formar un modelo

acoplado. El modelo acoplado es el que trabaja con la estructura y el comportamiento de dicho modelo.

3.3.1 Modelo atómico SISO

Decimos que un modelo DEVS atómico es SISO (single-in single-out) cuando nos encontramos con un sistema que tiene un evento de entrada y otro de salida. En la Figura 3.3.1-1.

Figura 3.3.1-1 Comportamiento Entrada/Salida DEVS

Los sistemas DEVS nos van a permitir realizar cualquier sistema en un intervalo de tiempo dado, experimentando un número finito de eventos. La entrada será una secuencia ordenada de eventos y su función de transición tiene una forma especial que limita la trayectoria para que la salida sea también una secuencia de eventos.

Un modelo DEVS atómico con un puerto de entrada y otro de salida (SISO), se representa de la siguiente manera:

$$DEVS = <X, S, Y, \delta i_{nt}, \delta_{ext}, \lambda, t_a>$$

Donde:

- X: Representa al conjunto de todos los posibles valores que un evento de entrada puede tener.
- S: Representa el conjunto de los posibles estados secuenciales. DEVS consiste en una secuencia ordenada de estados pertenecientes a S.
- Y: Conjunto de posibles valores de salida.
- δi_{nt}: (S->S): Función que representa la transición interna, describe la transición de un estado al siguiente estado secuencial de S.
- δ_{ext}: (Q x X -> S): Función que representa la transición externa. Es importante tener en cuenta que va a tener dos argumentos de entrada: un elemento del conjunto Q y otro del conjunto X. Devolverá un elemento del conjunto S.
  El conjunto Q, lo definiremos de la siguiente forma:
  $$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$$
  Siendo *s* un elemento de S y e un número real positivo (haciendo referencia al tiempo transcurrido desde la anterior transición del estado). El valor siempre estará acotado:
  $0 \leq e \leq ta(s)$
- λ: S->Y : Función de salida. El argumento de entrada pertence al conjunto S y devolverá un elemento del conjunto Y.

- ta: Función de avance de tiempo. El argumento de entrada vuelve a ser un elemento de S, que lo llamaremos s. La salida devuelve un número real positivo, desde 0 a infinito (ambos inclusive). Este número indica el tiempo que permanecerá en el estado s en ausencia de eventos de entrada.

La interpretación de estos elementos se realiza de la siguiente manera. Vamos a suponer que en un instante de tiempo t1, estaremos en el primer estado s1. En caso que no se produzca ningún evento de entrada, el sistema permanecerá en s1 durante ta(s1) hasta el instante t1+ta(s1) :

- Si ta(s1) es cero, entonces seguiremos estando en el estado s1, por lo tanto, no podrán intervenir ningún tipo de eventos de entrada. S1 será denominado como **estado transitorio.**
- Si ta(s1) es infinito, el sistema permanecerá en s1 hasta que ocurra un evento de entrada. En este caso, s1 será conocido como **estado pasivo.**

Llegando al instante en el que t2=t1 + ta(s1), se produce una transición interna. Es decir, se realizan las siguientes acciones según el orden que se sigue a continuación:

1. Asignamos a *e* el tiempo transcurrido desde la última transición: e=ta(s1).
2. La salida cambia de valor y1 = λ(s1).
3. El estado del sistema pasa de s1 a s2. S2= δi_{nt}(s1).
4. Planificamos la siguiente transición interna para t2 + ta(s2). Añadiremos la transición a la lista de eventos.

Si antes del instante en el que está planificada la siguiente acción, llega un evento de entrada, entonces en ese instante en el que llega el evento de entrada, se produce la transición externa.

Aplicando una transición a nuestro ejemplo (ya sea interna o externa), pasaremos al estado s3 en el instante t3 y que en el instante t4 llegará un evento de entrada de valor x1. Nos aseguramos que ta(s3) > e = t4 – t3.

En el caso en el que sea menos o igual, se habría producido una transición interna antes de la llegada del evento externo.

Cuando llega el evento externo t4, se van a producir las acciones anteriores:

- El estado del sistema cambia al instante debido al evento de entrada. El nuevo estado viene dado de evaluar la función de transición externa con tres argumentos: el evento de entrada, el estado y el tiempo transcurrido desde la última transición:
  s4 = δext(s3, e, x1), donde e = t4 − t3.

- Eliminaremos el evento interno de la lista de eventos que estaba planificado para el instante t3 + ta(s3) y en su lugar incluiremos el instante para el estado cuarto: t4 + ta(s4). Si no se produce ningún tipo de evento externo, ese instante se corresponderá con el instante en el que se produzca la próxima transición interna

Durante la transición externa, no se puede producir ningún evento de salida. Los eventos de salida se producen justo antes de las transiciones internas.

Esta explicación hace referencia al funcionamiento del modelo DEVS que sugiere cómo podría ser la operación de un simulador que ejecute este tipo de modelos para generar su comportamiento.

A continuación, pasaremos a explicar un ejemplo que hemos realizado en el que se verá el funcionamiento de un modelo atómico. Dicho ejemplo, trata sobre el funcionamiento que tendrá una máquina de bebidas:

3.3.2 Máquina expendedora

Consideremos una máquina simple que solo nos va a proporcionar bebidas, en este caso Coca Cola o Pepsi. Tendremos tres eventos de entrada que serán elegidos por el usuario. Estos botones pueden ser: "Introducir moneda" o marcar la bebida que desee, ya sea Coca Cola o Pepsi. También tendremos tres salidas, que serán las mismas según el estado al que se haya ido cambiando. Para nuestro ejemplo, dispondremos de una máquina de estados que nos va a facilitar la comprensión de nuestro ejemplo:

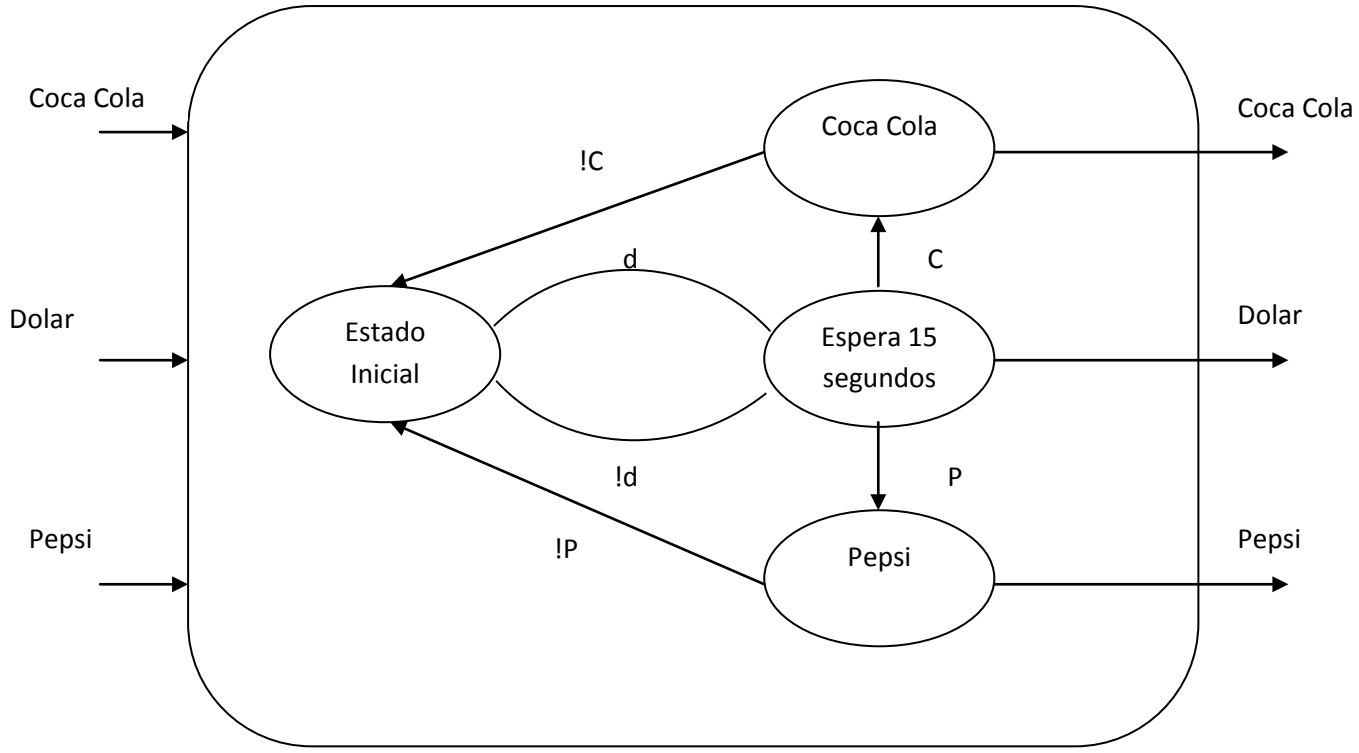


Figura 3.3.1-1Máquina de estados de Máquina Expendedora

Para realizar este ejemplo, vamos a tener tres módulos:

- **MaquinaCocaCola:** Nos facilitará una interfaz a través de la cual podemos insertar moneda en la máquina o elegir bebida

- **Planta:** Va a ser quien se encargue de gestionar la máquina de estados mediante eventos. De tal modo que controlará los tiempos de cada uno y dirigirá a que estado tienen que pasar.

- **Controlador:** Se encargará de gestionar las peticiones que se hagan a través de la referencia, por ejemplo si se agota alguna de las bebidas, nos lanzará un mensaje y no dejará que continuemos.

Las posibles elecciones de la máquina, (explicadas posteriormente), serán los eventos que generen transiciones en la máquina de estados. Para cada estado, tendremos un tiempo (ta) (como el explicado anteriormente), que será el tiempo máximo que permanecerá la máquina en ese estado sin que reciba ningún evento. Pasado este tiempo la máquina cambiará al estado que se le indique e caso de no recibir nada. O, en caso de que el tiempo de espera ta sea infinito, permanecerá en ese estado hasta recibir un evento.

Según los eventos que se reciban en los distintos estados, el sistema tendrá un comportamiento u otro (como el que tendría una máquina expendedora). Según este comportamiento, tendremos distintas salidas para nuestro sistema de eventos, dependiendo de los eventos de entrada.

3.3.2.1 Módulo MaquinaCocacola

Este módulo es básicamente una interfaz a través de la cual haremos nuestras elecciones. Cuando se arranque el servicio podremos ver lo siguiente:

Figura 3.3.1.1-1Interfaz Referencia

Donde podremos seleccionar los siguientes botones:

- Insertar Moneda: Si pulsamos en este botón el servicio referencia recibirá como entrada una “d”, de dólar, que pasará al controlador y esto pondrá en marcha la planta.
- Botón Cocacola: Una vez hayamos introducido el dólar, podemos seleccionar este botón para pedir una cocacola
- Botón Pepsi: Igual que el de cocacola pero para pedir pepsi.
- Boton Ok: Una vez hayamos pulsado nuestra elección de bebida, ésta no se gestionará hasta que no pulsemos este botón, que será la manera de confirmar el pedido y que el valos “c” o “p”, para cocacola y pepsi respectivamente, se envíen al controlador.
  Si pulsamos este botón sin haber seleccionado ninguna bebida nos saldrá un mensaje de aviso.

En este servicio tendremos un método: “UpdateReferencia”, que actualizará el estado del servicio, y hará que se envíe una notificación al servicio Controlador, que tendrá declarado a la Máquina como partner.

Como variables de estado tendremos la variable elección, que almacenará una letra que indique qué es lo último que hemos pedido.

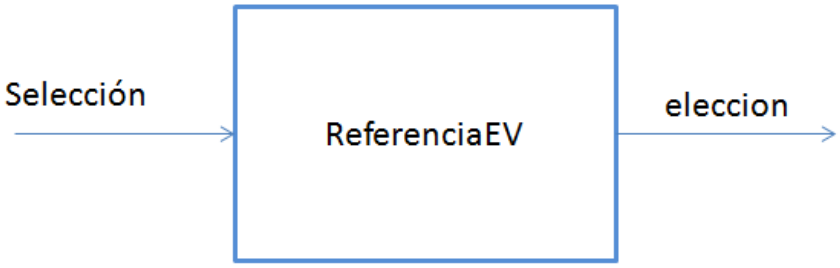


Figura 3.3.1.1-2Comportamiento ReferenciaEV

3.3.2.2 Módulo Controlador:

Este módulo se encarga de gestionar las peticiones que nos vienen de la máquina. Al iniciar el sistema, el servicio dispone de un número limitado de cocacolas y de pepsis, y a medida que se van pidiendo, este número disminuye su valor en uno. Si en algún momento, se hace una petición de una bebida que esté agotada, el controlador se encargará de lanzar un mensaje informándonos de ello. Una vez ha gestionado las peticiones, realizará una operación "UpdateControlador", para enviar las notificaciones a la planta, que tendrá a este servicio como socio declarado.

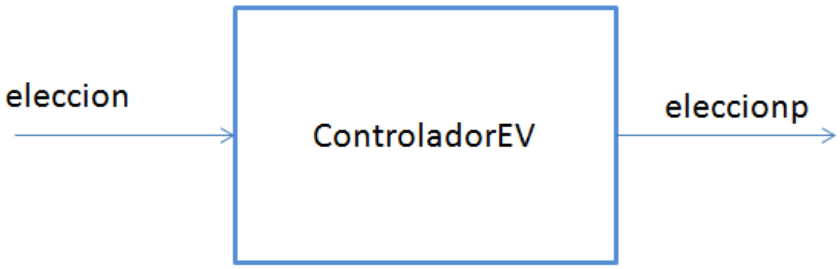


Figura 3.3.1.2-1 Comportamiento Controlador

3.3.2.3 Modulo Planta

El modulo planta va a ser quien se encargue de manejar la máquina de estados en función de la entrada que tenga. Para ello, vamos a utilizar varios temporizadores (timers) en los estados, que van a poseer un tiempo en el que van a poder realizar las acciones que llevan asociadas. Si estas acciones se cumplen, se cambiará de estado, y para ello se generará un evento que será el que se encargue de ello.

Dichos eventos son generados a través de las propiedades del temporizador, los cuales, como ya hemos dicho, van a ser los que se van a encargar del cambio de estado. Estos eventos son generados cada cierto periodo de tiempo, y nosotros decidimos, mediante las funciones que nos aporta dicho temporizador, si queremos que se repita una o más veces.

Para manejar nuestro temporizador, es importante conocer las funciones que hemos utilizado. En primer lugar, debemos crearlo mediante la llamada a su constructora:

```
System.Timers.Timer aTimer = new System.Timers.Timer();
```

El componente Timer es un temporizador basado en servidor que permite especificar un intervalo recurrente en el que se provoca el evento Elapsed en la aplicación. Va a ser a quien haga referencia ta dentro del sistema DEVS, ya que indica el tiempo que va a permanecer dentro del estado en ausencia de eventos de entrada. Se puede controlar este evento para realizar el procesamiento que se necesite. Se supone que el servicio recopila las instrucciones de envío de la bebida pedida, en vez de procesar cada pedido individualmente. Se puede utilizar Timer para iniciar el procesamiento de cada bebida cada 15 segundos (ta=15) y cambiar de estado posteriormente, de ello se encarga la siguiente instrucción:

```
aTimer.Elapsed += new ElapsedEventHandler(OnTimedEvent1);
```

Sin embargo, cuando la propiedad AutoReset se establece en false, Timer provoca el evento Elapsed sólo una vez después de que haya transcurrido el primer intervalo, que es lo que queremos que ocurra una vez que hemos insertado la moneda o cuando esperamos a que la máquina nos devuelva la bebida:

```
aTimer.AutoReset = false;
aTimer.Enabled = true;
```

Si la propiedad Enabled se establece en true y AutoReset se establece en false, Timer provoca el evento Elapsed sólo una vez, la primera vez que transcurre el intervalo.

De esta manera, conseguimos que una vez que el usuario haya introducido la moneda y haya seleccionado su bebida deseada, la recoja satisfactoriamente y la maquina quede preparada para un nuevo cliente que tendrá que repetir este proceso de nuevo.

3.3.3 Modelo acoplado: Juego de cartas

Para mostrar el comportamiento de un sistema acoplado, hemos creado un proyecto, compuesto por dos sistemas atómicos que interactúan entre sí formando así un sistema acoplado.

Como ejemplo para nuestro caso hemos elegido un juego de cartas sencillo: Cada partida consta de dos jugadores que comenzarán cada uno con n cartas y tendrán como objetivo quedarse sin cartas antes que su adversario. En cada turno, ambos jugadores sacarán una carta al azar y ganará el turno el jugador que obtenga la carta más alta. Al terminar la jugada, el ganador dará una de sus cartas al contrincante. En caso de sacar el mismo número, el ganador será el que haya sacado primero. (Al comienzo de la partida el primer jugador en sacar será el jugador1 por defecto, y en el resto de turnos, sacará primero el que haya ganado en el turno anterior).

Para simular este juego hemos creado dos servicios, (Jugador1, Jugador2). Cada uno de contendrá con una máquina de estados como la siguiente, (las dos se ejecutarán simultáneamente).

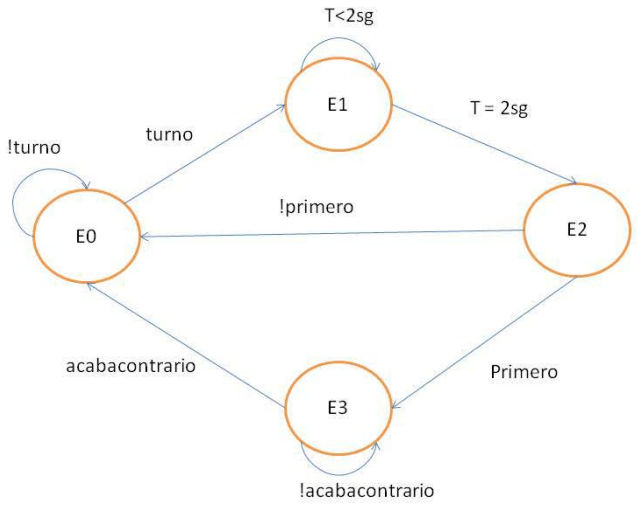


Figura 3.3.2-1Máquina de estados jugador

Explicación de la máquina de estados:

Cada estado de la máquina, tendrá un tiempo de espera asignado (ta), que le marcará a la máquina el tiempo que debe permanecer en dicho estado sin recibir eventos. En caso de que este tiempo se agote, y no se haya recibido nada, la máquina transitará a al estado que se le defina.

- **Estado 0:** Estado inicial para las dos máquinas. La transición de este estado al estado 1 dependerá de la variable "turno", que tomará valor "true", cuando sea el turno del jugador correspondiente, y permanecerá a "false" cuando el turno sea del otro jugador. En este caso, el ta es infinito, la máquina permanecerá en el mientras n sea el turno del jugador.
- **Estado 1:** Cuando sea el turno de un jugador, su estado pasará a ser el 1, donde permanecerá durante dos segundos de espera, y a continuación, pasará al estado 2 directamente. Para este estado, el ta es 2, y se ignorarán todos los eventos que se produzcan en ese intervalo de tiempo.
- **Estado 2:** Durante el estado 2, en primer lugar se generará el número aleatorio que corresponderá a la carta que saca cada jugador. A continuación el valor de este número será almacenado en el propio servicio y enviado al servicio del jugador contrario para que éste pueda disponer de él.
  Desde el estado 2, pueden producirse dos transiciones distintas, dependiendo de la variable "primero" de cada servicio. Esta variable nos indicará si el servicio

ha sido quien ha iniciado la jugada. Si el servicio fue el que tuvo el turno primero, la variable estará a “true”, y del estado 2, se realizará una transición al estado 3 y se cambiará la variable turno del servicio a false, y se pondrá a true en el otro servicio, para que éste comience a jugar. En el caso en que el servicio haya empezado en segundo lugar, la variable primero estará a “false”, y en este caso, antes de realizar la transición, se han de realizar los cálculos que determinen quién ha sido el ganador del turno. Para ello, el servicio comparará el número que obtuvo el otro jugador (almacenado en la variable numerocont), con el número que acaba de generar él mismo. Una vez sepa quién es el ganador, el servicio que está actuando actualizará las variables de turno, y primero, en su propio estado y en el de su contrincante, poniendo estas variables a true en el ganador y a false en el caso contrario. También actualizará la variable que lleva la cuenta del número de cartas de cada jugador, incrementando en uno el número de cartas del jugador perdedor, y restando una al ganador.

Una vez hecho todo esto, el servicio le indicará a su servicio oponente que ya ha terminado, y se realizará una transición al estado cero.

En caso de que el número de cartas de alguno de los jugadores se haga cero, se pararán las dos máquinas de estados y se mostrará un mensaje que nos indicará quién ha sido el ganador:

Figura 3.3.2.1 Final del juego de cartas

- **Estado 3:** El estado 3, es el estado de espera en el que permanecerá el servicio que haya empezado la jugada hasta que el otro servicio le indique que ha terminado, mediante la variable “acaba”, que indica si el otro jugador ha finalizado su jugada o no. Cuando esta variable se ponga a “true”, el estado hará una transición al estado cero y volverá a empezar el ciclo otra vez. En el estado tres, volvemos a tener un tiempo de espera infinito, hasta que no se reciba el evento que nos indique que el otro jugador ha terminado su jugada, la máquina permanecerá en este estado.

Los intercambios de información de ambos servicios se realizarán mediante cuatro operaciones de tipo Update (actualización de estado), declaradas en cada uno de ellos. En ellas cada servicio actualizará su estado con los valores que le correspondan, y enviará una notificación al servicio contrario informándole de los cambios.

- UpdatejValores: Actualizará las variables que hacen referencia a los números generados en cada turno (numerogen, numerocont).
- UpdateTurnos: Actualizará las variables turno y primero de cada servicio al final de una jugada, según este haya sido el ganador o perdedor de la misma.
- UpdateCartas: Actualiza las variables (cartas1,cartas2), incrementando o decrementando su valor según le haya ido al jugador en el turno
- UpdateFinPartida: En caso de que alguno de los jugadores se quede sin cartas, actualizará la variable finPartida en ambos servicios para que se paren las dos máquinas de estados.

4. Simulación usando entorno físico

A lo largo de este capítulo se va a explicar un nuevo ejemplo, con la novedad de añadir un módulo Simulador[14]. En este módulo haremos uso del VSE (Visual Simulation Environment), que como se mencionó en el primer capítulo viene incorporado en el MRDS.

En el siguiente apartado se explica de forma breve este entorno de simulación y las propiedades más importantes, o aquellas que nos van a ser de más utilidad.

4.1 Visual Simulation Environment (VSE)

El componente principal de la simulación es el motor de simulación, o en el caso de MRDS, el motor de simulación es un servicio. Para crear una simulación, primero se debe crear un servicio DSS y este utilizará el motor de simulación como un socio (partner). Con la declaración de un socio Simulación, se hace uso del servicio Simulation Engine, para lo cual, debemos añadir algunos using y referencias. Algunas de estas, con las razones por las que son necesarias, son:

- **Robotics Common:** Para utilizar: vector, matriz, color, postura, Drive, WebCam, Motor…
- **Simulation Common**: Entity, EntityState, SimulationState y las shapes (boxshape…)
- **SimulationEngine:** Inicialize, actualize, render, insert, remove, state… para las entidades.
- **SimulationEngine.Proxy**: Necesario para acceder a los elementos de la simulación Engine como los contracts y los puertos.
- P**hysicsEngine**: Representa la 'physics engine scene' que contiene todas las entidades de la simulación. Hay que añadirlo para usar los métodos de la physicsEntity.
- **Microsoft.Xna.Framework.Dll:** contiene el código de Rendering.
- **Xna.Framework**: tipos básicos como matriz o vector.
- **Wna.Framework.Graphics**:effect, texture…

Para crear el entorno de la simulación que se desea simular, hay dos formas de hacerlo, mediante código o mediante un manifiesto .xml.

Mediante manifiesto, con el VSE[15], se hace de forma más gráfica, se puede crear la escena y las entidades que se desean a través de un editor, y guardar la escena creada o modificarla. Al guardarla, se crean dos archivos XML. Por ejemplo, si se implementa un programa llamado Ej se crean los archivos, *Ej.xml* que contiene la simulación, el estado de las entidades; y *el Ej.manifest.xml* que contiene todos los servicios

necesarios. Estos archivos se pueden cargar luego en el simulador o añadirlos en el proyecto de visual para que formen parte de nuestro proyecto.

Mediante código, resultará más sencillo acceder a las entidades para realizar modificaciones en ellas, como modificar la velocidad, la posición, la orientación o aplicar algún tipo de fuerza.

En este apartado se explica de forma breve las principales funcionalidades del editor VSE, concretando más aquellas que se usan en el ejemplo explicado en el Apartado 4.2.

4.1.1 Editor del VSE

- **Modos de uso**, el RUN y el EDIT.

  El modo de ejecución, ejecuta la simulación pero sin permitir cambiar propiedades.

  El modo de edición, permite tanto añadir nuevas entidades como modificar las propiedades de las ya existentes. Permite también cargar y guardar entidades como código XML.

- Formas de representar las entidades (**Render**).
  - Visual: representa la vista en 3D.
  - Wireframe: Representa como una estructura metálica.
  - Física: muestra los contornos físicos.
  - Combinado:visual + física.

- **Physics**: Permite activar o desactivar las fuerzas físicas, con el Enable. O modificar los ajustes como la velocidad de simulación. (Se puede ajustar la cámara para que se pueda usar para golpear objetos.).

4.1.2 Entidades

Una entidad[16] es un objeto físico que se usa en una simulación. Puede ser tanto dinámico como estático. Estas entidades tienen ciertas propiedades según el tipo de entidad que sea y se pueden modificar estando en el modo Edición del simulador o mediante código si se tienen definidas las entidades programáticamente en lugar de con .xml.

Estando en el editor, también se puede seleccionar la entidad en el menú de la izquierda o presionando con el botón derecho encima del objeto, abrir su panel de propiedades y modificar directamente a los valores que nos interesan. Será al volver al modo ejecución cuando se vean aplicados los cambios.

Si realizamos los cambios programáticamente, lo haremos haciendo referencia a la entidad, declarada y según el tipo de propiedad, con State, PhisicsEntity…, según se define a continuación.

Tipos de entidades:

- **Dinámica:** está bajo el control de la física. Su posición y movimiento son resultado de la gravedad y las colisiones con otros objetos.
- **Cinemática:** otros objetos chocarán pero éste no cambia su posición a menos que se cambie manualmente. *La rampa es un buen ejemplo, si fuese dinámica, caería al suelo por no tener ningún apoyo.*
- **Estática:** no tienen masa, se componen de formas que tienen masa cero. Estas entidades pueden colisionar con otras pero no se mueven, están destinadas a permanecer en la misma posición durante toda la simulación.

Propiedades:

- **EntityState:** se asocia a otro cuadro con otras propiedades para definir como aparece y se comporta la entidad.
- **ServiceContract**: para incluir la ruta asociada a un archivo HTML del contrato de un servicio que nos interesa asociar, como puede ser algún drive.
- **InitError:** Explica porque ocurrió un error. La entidad se muestra con un punto rojo.
- **Flags:** indica como es representada la entidad.

  None

  UseAlphaBlending: permite transparencia.

  DisableRendering: existe, pero no es visible.

  InitializedWithstate: la entidad puede ser construida modificando el xml asociado a la escena de simulación.

  DoCompletePhysicsShapeUpdate: fuerza una actualización de todas las formas físicas que forman parte de la entidad.

  Ground: si la entidad es usada como entidad de suelo.

- **ParentJoint:** Específica su relación padre/hijo si la tiene.
- **Position/Rotation**
- **Meshes:** define las mallas asociadas a la entidad. Una malla que se utiliza para darles una apariencia más detallada y que se vean más realistas. Forma parte de la colección VisualEntityMesh. Se usan materiales para formar el físico del objeto. Pueden estar compuesto de más de un material.
- **MeshRotation:** normalmente vale 0.
- **MeshTranslation:** define las coordenadas que controlan como es trasladada la malla al origen de la entidad, normalmente vale 0.
- **MeshScale:** indica la escala de la malla, que normalmente valdrá 1.

- **Shapes:** depende del tipo de entidad, si es una SingleShape como cilindro, esfera… se pueden definir propiedades adicionales con esta opción, en un cuadro diferente.

Dos objetos no pueden ocupar la misma posición, si hacemos una copia de un objeto, al tener la misma posición no se apreciará la copia hasta que se cambie al modo Run, donde el objeto se reproducirá chocando con el objeto original.

Materiales:

Se usan para contrastar o diferenciar entidades entre sí. Una entidad puede tener varios materiales, así pues tendremos una lista de materiales con distintas mallas. Se pueden modificar directamente los valores o con el cuadro de colores. Básicamente, cambia el color de la entidad.

Textures: Se usan archivos .dss como texturas. Se asocian a los objetos para simular una textura.

Propiedades del State: que determinan como se ve la entidad.

Para cambiar datos como la masa, hay que abrir el nuevo cuadro de propiedades presionando los tres puntos de apartado EntityState.

El estado (EntityState) de la entidad contiene propiedades que afectan a la apariencia visual de la entidad así como su comportamiento físico.

- **DefaultTexture:** si no hay una malla especificada, se genera la malla por defecto de la forma. Soporta .dds,.bmp, .jpg,.png,.tif
- **Effect:** Ésta es la propiedad utilizada para renderizar la entidad. En muchos casos se puede usar la que viene por defecto (SimpleVisualizer.fx) o crear una específica e insertarla aquí.
- **Mesh:** si hay una malla especificada aquí, es usada para representar toda la entidad. Si no hay ninguna especificada, se usa una por defecto. Deben ser archivos .obj o .bos.
- **Flags:** Definen algunos comportamientos característicos de la entidad. Si no hay ninguna definida, esta Dynamic, que significa que la entidad se comporta como una entidad dinámica normal controlada completamente por el comportamiento físico.
    - **Kinematic:** Si esta activada, significa que la entidad es cinemática, es decir, que si posición y orientación es controlada por un programa externo. Esta entidad no es influida por la gravedad o por la colisión de otros objetos, pero sí que puede influir en otros al colisionar con ellos.
    - **IgnoreGravity:** la gravedad no influye en esta entidad.
    - **DisableRotation/X/Y/Z:** no habrá rotación en el eje especificado. No tiene efecto en cinemáticas.

- **Name:** nombre de la entidad, debe ser ÚNICO.
- **Velocity:** Este vector mantiene la magnitud y dirección de la velocidad.
- **AngularDamping/LinearDamping**: Estos coeficientes de amortiguamiento son números que van de cero a infinito. Ellos proporcionan un medio para modelar los efectos del mundo real tales como la fricción del aire, que tienden a ralentizar el movimiento lineal y angular de un cuerpo. Cuanto mayor sea el número, más el movimiento lineal o angular ralentiza. *Por ejemplo, las entidades con el valor predeterminado de cero (0) para amortiguación continuará rodando por el suelo hasta que alguna otra fuerza se lo impida.*
- **Mass/Density:** Estas propiedades son en realidad dos formas de especificar la misma cosa. Si la masa es distinto de cero para una o más de las formas de física en una entidad, el motor de física se calcula la masa total y la densidad de la entidad a partir de la suma de la masa de las formas. En este caso, estos dos valores se ignoran cuando la entidad se inicializa. Si la suma de las masas de las formas es cero, entonces el motor de física utiliza uno de estos valores para determinar la masa de la entidad. El valor de la masa proporciona un método abreviado para especificar la masa total de una entidad. El motor de física se calcula el centro de masa suponiendo que la densidad de las formas de la entidad es uniforme. El valor de densidad proporciona otro atajo para especificar la masa total. Si la densidad se especifica, la física del motor calcula la masa total basado en el volumen de las formas en la entidad. De nuevo, el motor se supone que la densidad es uniforme en toda la entidad.
- **InertiaTensor:** Esta propiedad especifica la distribución de la masa en la entidad. Es decir, como es de difícil rotar la entidad según el eje. Por ejemplo un cilindro rota muy bien en el eje longitudinal pero más difícil por otro eje. Si no se especifica se calcula según la posición y tamaño de las formas.

Entidades para crear el suelo y el cielo del entorno:

El simulador proporciona dos entidades integradas para definir el cielo, así como una serie de entidades para definir diferentes tipos de terreno. Se puede construir una escena sin estas entidades, pero para que el ambiente parezca más realista, sobre todo para las escenas al aire libre es recomendable hacer uso de éstas. Solo mencionar los dos tipos de entidades, y alguna de sus propiedades, en el ejemplo explicado más adelante en el capítulo, se mostrará el código necesario para insertar cada una de ellas.

SkyEntity y SkyDomeEntity

La primera de ellas, es una entidad que rodea toda la escena con una malla esférica 3D. Con esta entidad en realidad, la mitad de la malla insertada no se ve nunca, pues esta tapada por la entidad suelo. La segunda de las nombradas, es más eficaz, ya que

utiliza media cúpula para rodear solo la parte por encima del suelo. Aun siendo ésta más eficaz, para simulaciones de gran altitud es preferible el uso de la primera, pues sino se apreciaría un espacio entre la tierra y el cielo.

Ambas de ellas constan de las propiedades de textura, utilizando archivos .dds; ambas pueden agregar luz difusa a la escena, o modificar la intensidad y el color de la luz sobre las entidades definidas. También se puede modificar el nivel de niebla de la escena, tanto el color como la distancia a la que se encuentra.

HeightFieldEntity

Estas entidades se utilizan para crear suelos planos. Es una de las entidades más importantes porque sin ella, los objetos caen al abismo. Esta entidad tiene elevación cero. Una de sus propiedades principales es la textura, una imagen de tamaño un metro cuadrado, que se repite indefinidamente. Otras propiedades a destacar son la fricción y la restitución, que también pueden ser modificadas.

Existe también una entidad TerrainEntity para definir terrenos montañosos, con cambios de elevación.

Entidades para crear objetos más complejos:

SingleShapeEntity:

Esta es la forma más fácil de añadir un solo objeto para el medio ambiente. Decidimos la forma, ponemos las dimensiones y lo añadimos.

Tiene una serie de propiedades, divididas en MassDensity, Materiales, Varios y Posición.

MassDessity:

- AngularDamping/LinearDamping: lo mismo que en EntityState, pero se aplica sólo a una forma.
- Mass/Density
- CenterOfMass: posición del centro de la masa, que es el punto alrededor del cual gira la forma natural.
- InertiaTensor: lo mismo que en EntityState, pero se aplica sólo a una forma.

Material:

- DynamicFriction: indica la fricción de la forma con otra forma cuando está en movimiento.
- StaticFriction: indica la fricción de la forma con otra forma cuando está parado.
- Restitution: indica el rebote de la forma. Siendo 0 sin rebote.

- MaterialIndex/MateialName: indica el material usado.

Misc:

- DiffuseColor:color predeterminado para crear la malla.
- Dimensions/Ratio.
- EnableContactNotifications: se puede configurar para proporcionar una notificación cuando entran en contacto con otra forma. Útil con los servicios.
- Nombre
- ShapeId
- TextureFileName

Posicion:

- Position/Rotation: especifica el desplazamiento y orientación de la forma en comparación con el origen de la entidad.

MultipleShapeEntity: Igual que el anterior pero para múltiples formas.

SimplifiedConvexMeshEnvironmetEntity: deben ser entidades estáticas, actúan como si tuvieran masa infinita.

TriangleMeshEnvironmetEntity: Similar a la anterior. No podemos usar este siempre porque tiene limitaciones. La detección de colisiones no es tan buena.

4.1.3 Insertar la simulación en un proyecto.

Tal y como hemos hecho en los capítulos anteriores al declarar los socios, para que se ejecute la simulación, debemos en primer lugar definir las entidades que nos interesan, definir un socio SimulationEngine. Y al ser una simulación, tenemos que definir el entorno de simulación. Esto se explica más detalladamente en el apartado siguiente, en el módulo Planta (4.2.3) acompañado del ejemplo.

4.2 Ejemplo: Quatrirrotor

En este capítulo, se va a realizar la simulación del control de un quatrirrotor. Un quatrirrotor es un vehículo de vuelo que consta de cuatro rotores situados en forma de cruz, la distribución es simétrica y estos están emplazados en los extremos. Los movimientos se consiguen variando la velocidad relativa de cada motor.

El ejemplo que vamos a estudiar en este capítulo, consta de los mismos módulos que los ejemplos que hemos visto en los distintos sistemas. Así se vuelve a ver la eficiencia del sistema de módulos que hemos definido, y como se puede aprovechar los

módulos, insertando simplemente la funcionalidad especifica de este ejemplo sin modificar las conexiones. A excepción de la unión de un nuevo módulo, el Visualizador.

Así pues el modelo del ejemplo que se va a seguir es el ilustrado en la siguiente figura:

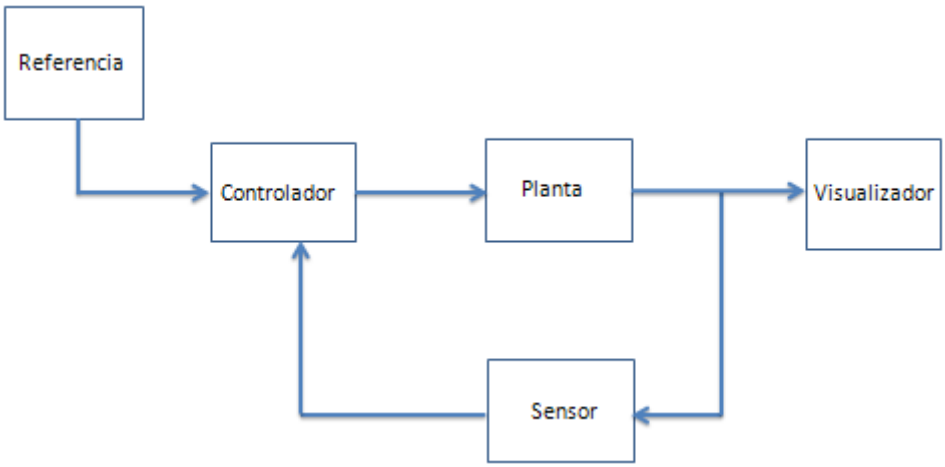


Figura 4.2-1 Módulos del ejemplo del quatrirrotor

Como se observa, el módulo principal es el servicio Controlador, quien define como socio a Referencia y a Planta, se suscribe a ellos teniendo así los puertos necesarios para su comunicación. Es el módulo planta quien tiene definido y suscrito el socio Sensor con el que intercambiará mensajes con los valores necesarios; y un socio motor de simulación para llevar a cabo la simulación deseada. El servicio sensor, tiene que tener declarado como socio el controlador, pues necesita la conexión de puertos para enviarle los valores correspondientes. No creará un nuevo servicio controlador, sino que se conectará a la instancia ya creada.

4.2.1 Módulo Referencia

Este módulo consta de un interfaz de usuario con el cual el usuario podrá seleccionar el modo con el que desea manejar el quatrirrotor, a poder seleccionar entre manual y automático. Esta interfaz se ha realizado con un WinForm, un elemento del Visual Studio. Es una herramienta gráfica que con la ayuda de un cuadro de herramientas y una tabla de propiedades hace su elaboración muy sencilla, pudiendo añadir elementos gráficos como botones, paneles, cuadros de texto… y definir una serie de propiedades para cada uno de los elementos así como eventos a los que reaccionar, como eventos de ratón o teclado.

Con el modo automático, como se observa en la figura 4.2.1 -1, se puede insertar la posición y/u orientación a la que el usuario desea vaya el quatrirrotor automáticamente.

En el modo manual, encontramos muchos más controles, pues se controla manualmente la fuerza que se aplica a cada uno de los motores. Se puede modificar a

cada motor de manera individual, en la parte izquierda, utilizando los botones '+' y '-', se incrementa o disminuye el valor 0.5 respectivamente al motor en el que se aplica. En la parte central, podemos insertar en el cuadro de texto el valor que deseamos añadir o quitar cada uno de los valores que tienen los motores en ese instante. En la parte derecha, podemos modificar los valores de los motores 1-3 o 2-4 según el tipo de movimiento que buscamos, si queremos el cabeceo del quatrirrotor utilizaremos entonces el primero de estos. Por último, en la parte inferior, se encuentra un botón para detener todos los motores.

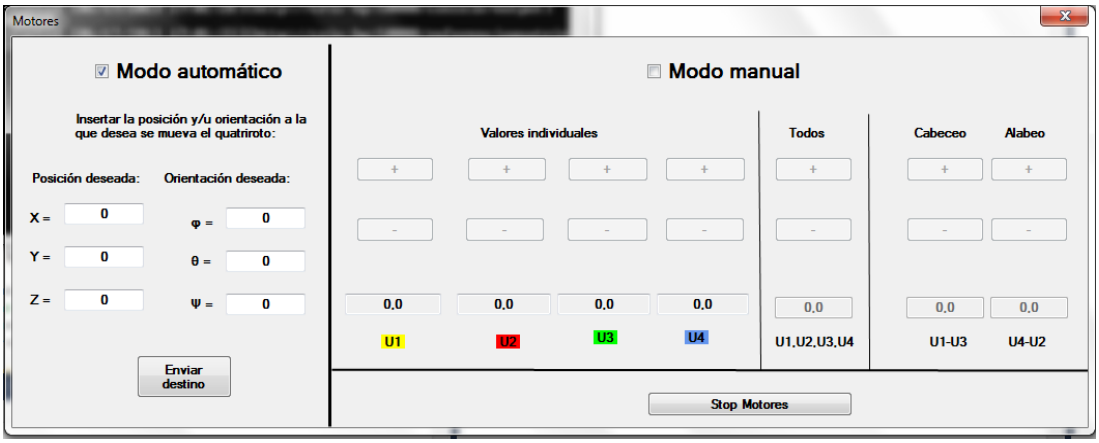


Figura 4.2.1-1 Referencia - Modo Manual.

Al presionar cada uno de los botones descritos, se envía por los puertos que se han descrito al suscribir este servicio con el controlado, un mensaje con los valores convenientes, en caso de presionar el botón 'Enviar' del modo automático, se envía la posición y la orientación deseada del usuario, en caso del modo manual, el valor de las fuerzas a aplicar a cada uno de los cuatro motores. Así, el comportamiento del módulo se observa en la siguiente figura:

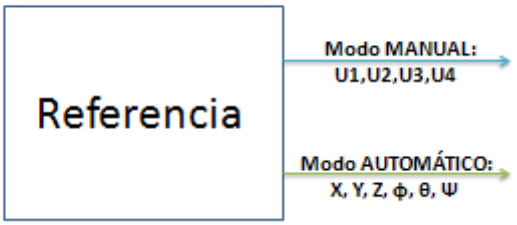


Figura 4.2.1-3 Comportamiento del módulo referencia.

4.2.2 Módulo Controlador

El módulo controlador es el servicio encargado de controlar el quatrirrotor cuando está en modo automático o simplemente pasar los valores de las fuerzas a aplicar a los motores que vienen de referencia si esta en modo manual. Así pues el controlador tendrá una variable de estado para poder saber en cada momento en que modo esta, y así saber cómo reaccionar ante los valores recibidos.

Este módulo, recibe valores de referencia y de sensor, así pues tiene estos definidos como socios. De sensor recibirá cada vez que se actualiza la entidad (cada 10ms aproximadamente), su posición y orientación actual. Como se observa en la figura 4.2.2-1, en caso de estar en modo manual, simplemente almacena dicha posición y orientación sin utilizarlas.

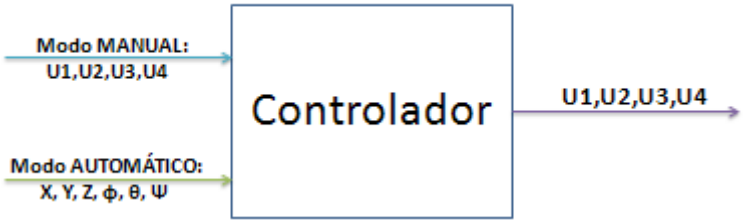


Figura 4.2.2-1 Comportamiento controlador.

En caso de estar en modo automático cada vez que se reciben valores de sensor, el controlador realizará las cuentas necesarias, los cálculos de los errores y todo lo que se necesita para calcular las nuevas fuerzas a aplicar a los motores.

Para realizar los cálculos, lo primero es tener colocados los motores de la siguiente manera:

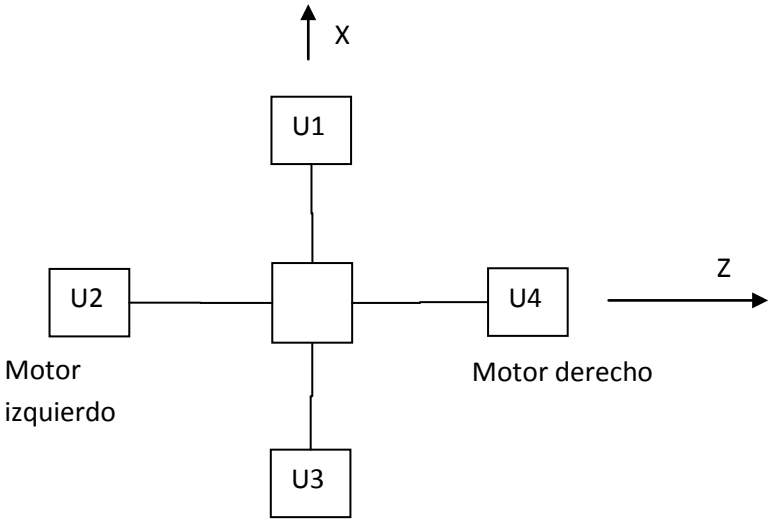


Figura 4.2.2-2 Colocación de los motores del quatrirrotor

Lo que se tienen que calcular son las fuerzas para cada motor, U1, U2, U3 y U4. Para ello, necesitamos calcular el producto de las siguientes matrices:

$$\begin{bmatrix} U_{motor1} \\ U_{motor2} \\ U_{motor3} \\ U_{motor4} \end{bmatrix} = \begin{pmatrix} 0 & 0.5 & 0.25 & 0.25 \\ -0.5 & 0 & -0.25 & 0.25 \\ 0 & -0.5 & 0.25 & 0.25 \\ 0.5 & 0 & -0.25 & 0.25 \end{pmatrix} \cdot \begin{bmatrix} U_{\phi} \\ U_{\theta} \\ U_{\psi} \\ U_{z} \end{bmatrix}$$

Figura 4.2.2-3 Matrices para el cálculo de fuerzas

Para calcular este producto, necesitamos el valor de las señales de control. Estas señales de control son las que se transforman en fuerza para los motores U1, U2, U3, U4. Como se observa en la figura anterior (4.2.2-3), si se desea ascender, únicamente se tiene la señal Uz que transfiere una tensión proporcional a cada uno de los cuatro motores. Si por otro lado nos interesa el movimiento de alabeo, la señal U_Φ actúa sobre los motores izquierdo y derecho (U2 y U4), aplicando una señal con la misma magnitud pero sentido contrario. De esta misma forma actúan los demás movimientos.

En el momento que se recibe un destino desde referencia, el modo del controlador cambia a Modo Automático, es entonces tras esto, o cuando se recibe una posición actual desde sensor y el modo es automático, cuando se llama a la función calcularFuerzas(). Para realizar este método, se le pasa por parámetro el tiempo, un vector de estados, que es un vector de valores acumulados Xv que se recalcula en cada llamada, y un vector Uin con los valores del quatrirrotor(posición, velocidad y orientación).

En este método, se realiza una llamada a un método (fgxutXYH()), que tiene como entradas, las explicadas anteriormente. Y obtiene como salida, un vector con el vector a acumular (Xv) y el valor de las señales de referencia de control del pitch (Upitch), del roll (Uroll) y del control de altura (Uthrust).
Esta función lo que hace es coger los valores de las variables Xv y Uin y colocarlos en la llamada a nuevas funciones. Los cálculos se hacen en las funciones:

1.- fgXY(Xr,X,Xx); (Xr -> posX de referencia, posX del quatrirrotor, Xx valores acumulados) devuelve en salida[0] Upitch y en salida[1] valores acumulados para después.

2.- fgXY(Zr,Z,Xz); (Zr -> posZ de referencia, posZ del quatrirrotor, Xz valores acumulados) devuelve en salida[0] Uroll y en salida[1] valores acumulados para siguiente ejecución.

3.- fgH(Hr,H,Xy); (Hr -> posH de referencia, posH del quatrirrotor, Xy valores acumulados) devuelve en salida[0] Uthrust y en salida[1] valores acumulados para siguiente ejecución.

Una vez que se tienen estos valores, con Uroll y Upitch se deben calcular Uϕ y Uθ. Para ello se usa la función fgxutRPY() que es similar a la fgxutXYH(), es decir, recoloca las variables y llama a las funciones de cálculo necesarias:

1.- outf = fgRP(PHIc, PHI, PHIdot, Xr); (PHIc ángulo Theta deseado, PHI ángulo theta del quatrirrotor, PHIdot velocidad de giro y Xr, variables acumuladas). Devuelve Uθ.

2.- fgRP(THEc, THE, THEdot, Xp); (THEc ángulo Theta deseado, THE ángulo theta del quatrirrotor, THEdot velocidad de giro y Xp, variables acumuladas). Devuelve Uϕ.

3.- Para el cálculo del control de Yaw , se aplicará el cálculo necesario directamente en la planta puesto que no existe una rotación real de las hélices, siguiendo la fórmula:

$$Torque = Kz *(U1 +U3 –U2 –U4)$$

A lo largo de los códigos descritos, se hace uso de una serie de constantes, previamente calculadas para los valores del quatrirrotor y según los cálculos de control necesarios.

4.2.3 Módulo Planta

Este módulo es el servicio que hace uso de todo lo explicado en el apartado 4.1. En este ejemplo se va a definir todo el entorno de simulación programáticamente para poder acceder a las entidades con facilidad y así poder consultar la posición y orientación, y lo más importante, aplicar una determinada fuerza.

Este módulo es el que vamos a utilizar como módulo principal, es decir, el que ejecutaremos y será él el encargado de arrancar el resto de módulos. Este recibe de controlador los valores de las fuerzas a aplicar a cada uno de los cuatro motores, y envía como salida al sensor la posición y orientación actual del quatrirrotor (Figura 4.2.3-1). Esta salida, es enviada cada 10 ms, pues la fuerza se aplica cada 10 ms.

Figura 4.2.3-1 Comportamiento del módulo planta.

Para crear la simulación, hemos definido un socio simulación, de la forma habitual. Y al ser una simulación, tenemos que definir el entorno de simulación.

Para ello, en el método Start (después de la instrucción base.Start()) debemos añadir el entorno. Para mayor simplicidad, en el ejemplo se ha añadido la llamada a un método PopulaWorl(), es este método el que llama a una serie de métodos, en nuestro caso addCamera(), addSky(), addGround() y addQuatrirotor(). Los tres primeros son básicos, pues insertan lo que llamamos el mundo de la simulación, el suelo y el cielo, y la cámara, nuestro punto de vista. El último se ajusta más al ejemplo concreto.

Para insertar el cielo se hace uso de la entidad cielo y la entidad suelo, ya explicadas en el apartado 4.1.2. Y para insertar el quatrirrotor, se han insertado una serie de SingleShapeEntitiy, creando relaciones de padre-hijo entre ellas para que se muevan como una sola figura. Resultando la siguiente imagen:

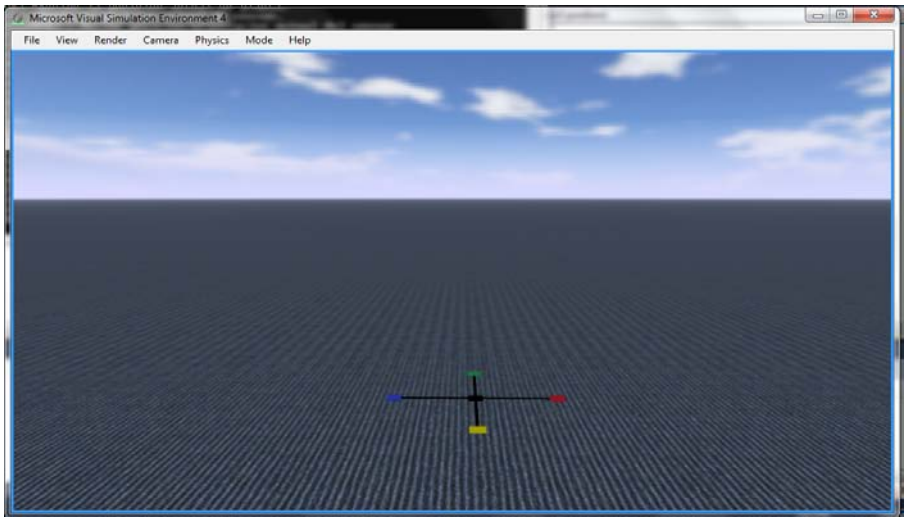


Figura 4.2.3-2 Entorno de simulación creado.

Puesto que lo que nos interesa es aplicar una fuerza repetidamente, se hace uso de un método periódico, es decir, un método que al terminar de ejecutarse hace una llamada recursiva a sí mismo, por lo que se está aplicando la fuerza permanentemente desde que se recibe el primer valor.

```
// Se llama periodica mente para que se mueva de forma continua
     void PeriodicTimerExample(DateTime dt)
     {
         int timeDelay = 10; //milisegunds

         ……

        // Wait for some time
         Activate(Arbiter.Receive(
             false,
             TimeoutPort(timeDelay),
             PeriodicTimerExample));
     }
```

Al recibir los valores se almacenan en variables del estado de la planta para que así el método periódico pueda acceder en todo momento al valor a aplicar a cada motor.

```
public virtual void plantaHandler(controlador.UpdateMovimiento
movimiento)
        {
            Console.Out.WriteLine("[PLANTA] - Recibo valores de
Controlador.");
            _state.U1 = movimiento.Body._valor1;
            _state.U2 = movimiento.Body._valor2;
            _state.U3 = movimiento.Body._valor3;
            _state.U4 = movimiento.Body._valor4;

        }
```

En este servicio, para poder aplicar la fuerza, es necesario añadirlo como tarea de la entidad, para ello, creamos una tarea, y la añadimos a la lista de tareas pendientes de la entidad siempre y cuando no haya tareas pendientes, pues sino se irían acumulando produciendo así una falsa simulación. Por lo que antes de añadir la tarea limpiamos la lista:

```
Se AÑADE TAREA→   disco1.DeferredTaskQueue.Clear();
                  disco1.DeferredTaskQueue.Post(new Task(tarea1));

TAREA→       public void tarea1()
             {
             float valor = (float)_state.U1;
             disco1.PhysicsEntity.ApplyForce(new Vector3(0, valor*k, 0), true);

             }
```

Esta tarea se realiza cada vez que se actualiza el entorno, y cada vez que ocurre esto cuando se envía a sensor la posición y orientación actual para que pueda realizarse el control del movimiento del quatriritor:

```
public void enviarPosicionActual(double posX, double posY, double posZ,
double orX, double orY, double orZ)
        {
            UpdateMovimiento mensaje = new UpdateMovimiento();
            mensaje.Body._posX = posX;
            mensaje.Body._posY = posY;
            mensaje.Body._posZ = posZ;
            mensaje.Body._orX = orX;
            mensaje.Body._orY = orY;
            mensaje.Body._orZ = orZ;
            _mainPort.Post(mensaje);
            Console.Out.WriteLine("[PLANTA] - Envio la pos/or actual de
planta.");

        }
```

4.2.4 Módulo Sensor

Para este ejemplo, el módulo sensor no es necesario, pero para seguir con el esquema del resto de ejemplos ya explicados se va a añadir con el único fin de recibir los valores de salida de Planta y enviárselos a Controlador.

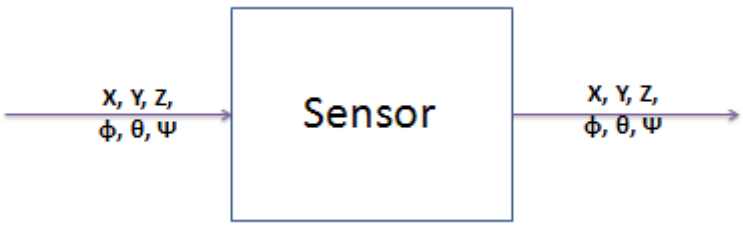


Figura 4.2.4-1 Comportamiento del módulo sensor.

4.2.5 Ejecución del ejemplo

Una vez se han creado los cuatro módulos, y se ha generado la solución de cada uno de ellos, se procede a ejecutarlo. Como hemos mencionado antes, el módulo principal es el servicio planta. Así pues, ejecutamos el servicio planta, desde Visual o desde la dss command, y veremos la salida de consola siguiente:

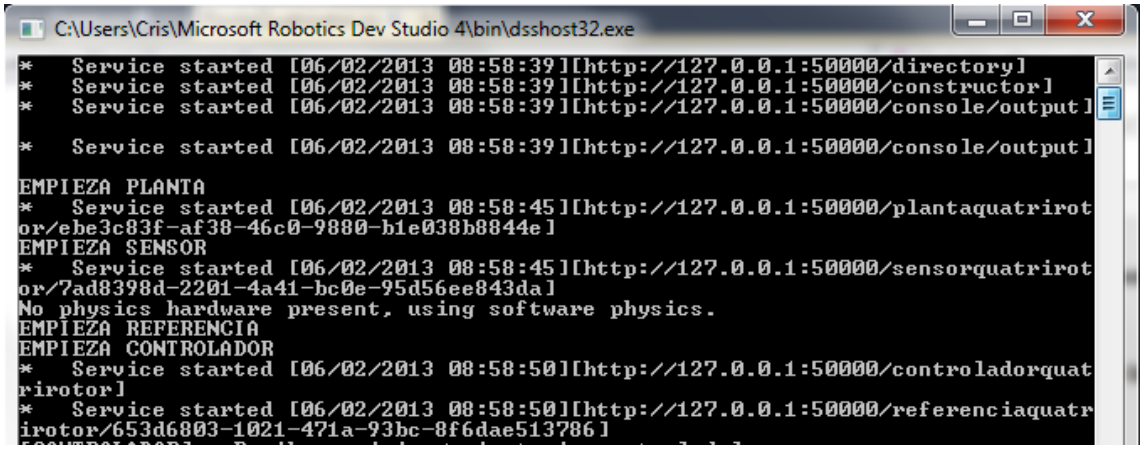


Figura 4.2.5 Salida de la ejecución del ejemplo.

En esta imagen se observa cómo se crean los demás módulos con solo haber ejecutado el módulo Planta.

5. Aplicaciones distribuidas

Escribir una aplicación utilizando DSS es simplemente crear las conexiones entre las entradas y salidas de los distintos servicios. Estos servicios pueden encontrarse en diferentes instancias de nodos DSS que pueden estar incluso distribuidos en diferentes equipos.

La posibilidad de crear aplicaciones distribuidas hace que el proyecto que llevamos a cabo amplíe su potencial. Pues el poder distribuir una aplicación, es decir ejecutar las distintas partes, los distintos módulos o servicios del proyecto en diferentes equipos, facilita a la hora de probar hardware antes de usarlo en un dispositivo robótico, o incrementa el rendimiento, pues se pueden distribuir las actividades costosas en diferentes equipos para aumentar el potencial de cálculo y por tanto el rendimiento.

5.1 Creación de aplicaciones distribuidas

Para crear las conexiones entre distintos nodos DSS, podemos realizarlas programáticamente o especificando la URL en los manifiestos. DSS ofrece unas herramientas que nos ayudaran a crear los manifiestos de los múltiples nodos y ejecutarlos.

Si lo que se desea es ejecutar los servicios en equipos o nodos diferentes, hay dos posibilidades, que se inicien cada uno de los servicios implicados manualmente, es decir, ejecutando el nodo y en él iniciar el servicio siguiendo el orden que corresponde a las dependencias, este caso se explica en el Apartado 5.3.1 y no requiere modificaciones de código. O se pueden realizar unas modificaciones en el código(esto se explica en el Apartado 5.2.1) y así que sean los propios servicios suscritos a socios los que inicien los servicios necesarios, teniendo así únicamente que iniciar los nodos DSS, evitando tener que iniciar cada servicio, esto se explica en el Apartado 5.3.2.

En ambos casos, para comprobar lo que se está ejecutando en cada nodo, y comprobar que se está ejecutando lo que queremos, podemos comprobar los nodos utilizados a través del navegador, esto se explica más adelante, en el apartado 5.6.

5.1.1 Ejemplo básico

A lo largo del capítulo, vamos a suponer un ejemplo sencillo con dos servicios, un servicio B que envía un mensaje al servicio que se suscriba a él. Y un servicio A que tiene definido como socio al servicio B. Como se ve en la siguiente figura, es el Servicio A quien crea la pareja de Servicio B, y por tanto quien arranca el servicio.

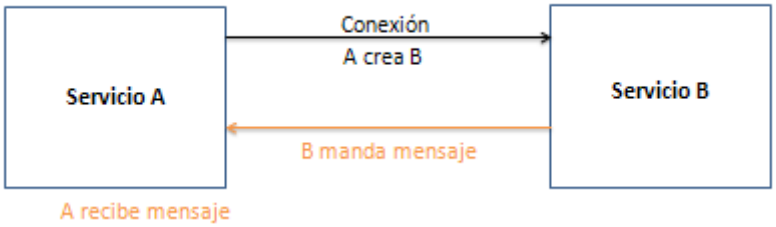


Figura 5.1.1-1 Ejemplo de conexión de los servicios A y B.

En la figura anterior se observa también como el Servicio B envía un mensaje al Servicio A y este lo recibe. El ejemplo consiste en ver por consola, como ejecutando el servicio A y el servicio B en distintos nodos (en el ejemplos también en diferentes ordenadores) el servicio A recibe el mensaje del servicio B. En la siguiente figura se ve el resultado que se obtiene por consola al ejecutar el ejemplo.

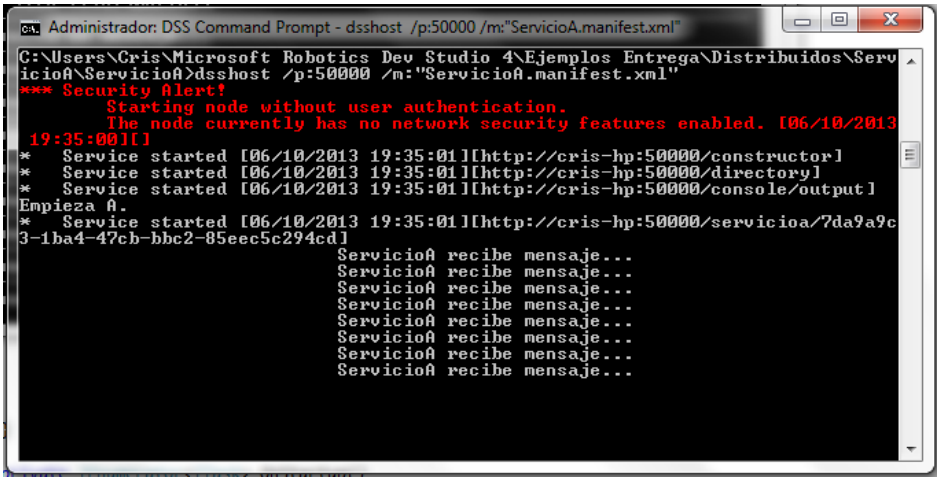


Figura 5.1.1-2 Captura de la consola Local con la salida de mensajes del ejemplo.

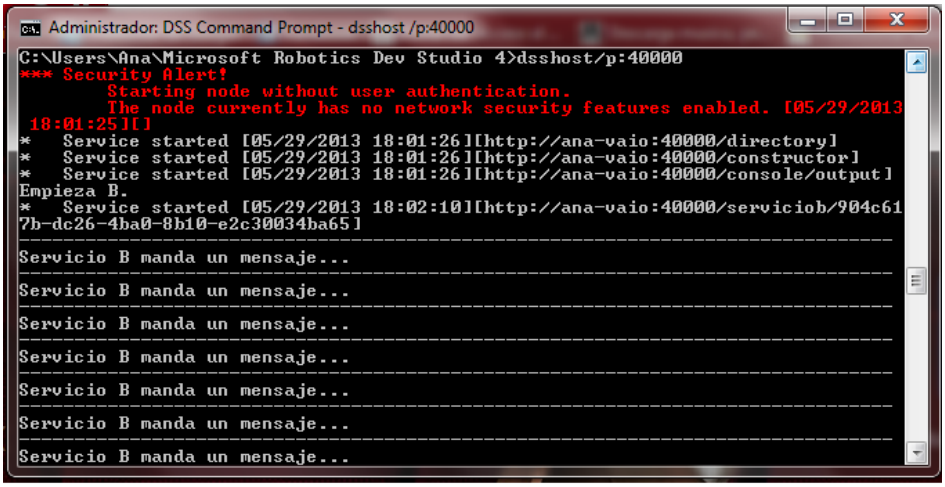


Figura 5.1.1-3 Captura de la consola remota con la salida de mensajes del ejemplo.

El primer paso es crear el servicio B de manera habitual. Explicamos a continuación como crear la declaración y subscripción del socio en el servicio A.

5.2 Aplicaciones distribuidas iniciando solo el servicio principal

Para ejecutar servicios en equipos diferentes, iniciándose de manera automática, es decir, en el ejemplo simplemente iniciando el ServicioA, siendo este quien inicia el ServicioB, es necesario suscribir los servicios de manera especial a la habitual(a la explicada en los apartados anteriores). Aunque hay que destacar que con la manera de declarar y realizar las conexiones que se explica en este capítulo, el proyecto funciona de igual forma si se quiere ejecutar en un único nodo. No siendo recomendable utilizar estos mecanismos, por ser éstos más complejos que los explicados en capítulos anteriores.

5.2.1 Modificaciones a realizar en el código.

En este apartado, se van a explicar las modificaciones que son necesarias realizar en el código[17], en la declaración de las "partners" y la suscripción entre ellas.

Para declarar un socio, hasta ahora hemos definido el nombre del servicio, el contrato del servicio creado y la politica de creación (UseExisting, UseExistingOrCreate…). Al iniciarse un servicio con socios declarados, este inicia los servicios declarados como parejas directamente, a menos que ya se esté ejecutando en el nodo actual.

Es posible utilizar un servicio que se ejecuta en otro nodo como un socio utilizando simplemente un manifiesto, pero no es posible iniciar un servicio de este modo. Para ello es necesario realizar unas modificaciones en el código. Así pues procedemos de esta manera, con el ejemplo descrito en el apartado anterior, a explicar las modificaciones a realizar.

La primera modificación es quitar la declaración de socio del servicio B. Manteniendo la declaración de los puertos (**1)** en la *Figura 5.2.1-1*). Esto significa que ya no se establece la pareja cuando se inicia el servicio. El servicio deberá establecer la propia asociación.

El segundo paso consiste en encontrar el nodo en el que se inicia el servicio. Esto se realiza mediante la especificación de una ruta a un directorio de servicios en su manifiesto.

Debemos añadir en el manifiesto la especificación del nombre y la ruta del nodo donde se ejecutará el servicio B. Esto se explica más adelante cuando creamos la aplicación distribuida con el Manifest Editor. Suponemos que llamamos al servicio B "Remote".

Esta información puede ser leída por el servicio en tiempo de ejecución llamando a base.FindPartner ("Remote"), el campo servicio en el objeto PartnerType devuelto contendrá el identificador uniforme de recursos (URI) especificado en el

manifiesto.Esta URI puede luego ser utilizada para crear un promotor del servicio usando base.ServiceForwarder<T>().

Para esto se requiere añadir dos declaraciones using para introducir los espacios de nombres Directory y Constructor que serán utilizados más tarde.

```
using ds = Microsoft.Dss.Services.Directory;
using cs = Microsoft.Dss.Services.Constructor;
```

A continuación, se muestra en una imagen la secuencia de pasos que debemos realizar.

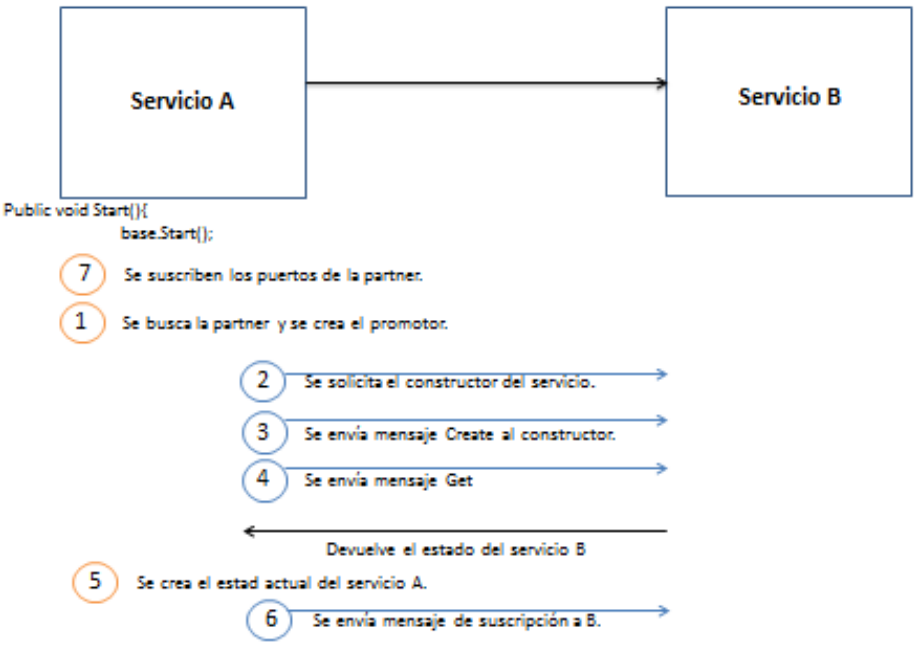


Figura 5.2.1-1 Pasos de las modificaciones.

En un método a parte o en el Start, después de la instrucción base.Start(), crearemos un promotor en el directorio remoto con el siguiente código(paso **1** de la *Figura 5.2.1-1*):

```
PartnerType distancia = FindPartner ("Remote");
ds.DirectoryPort RemoteDir = DirectoryPort;

if (distancia! = null &&! String.IsNullOrEmpty
(remote.Service))
{
RemoteDir = <ds.DirectoryPort> ServiceForwarder
(remote.Service);
}
```

Ahora que el servicio tiene un promotor en un directorio remoto de servicios, o en el caso de que falle el directorio local de servicios, puede consultar el directorio de una

instancia del constructor. Para ello añadimos a continuación el siguiente código (paso **2** de la *Figura 5.2.1-1*):

```
cs.ConstructorPort remoteConstructor = ConstructorPort;
ds.Query consulta = new ds.Query (
    new ds.QueryRequestType (
        new ServiceInfoType (cs.Contract.Identifier)
    )
);

remoteDir.Post (query);
yield return (Choice) query.ResponsePort;
ds.QueryResponseType queryRsp = query.ResponsePort;
if (queryRsp! = null)
{
    remoteConstructor = <cs.ConstructorPort>
ServiceForwarder (queryRsp.RecordList [0] Servicio.);
}
```

Este código envía un mensaje consulta al directorio del servicio para solicitar el constructor del servicio, identificado por su contrato. Si esto falla, entonces en él se utiliza el servicio constructor local.

El paso final es enviar un mensaje Create al constructor del servicio (paso **3** de la *Figura 5.2.1-1*).

```
string servicioBService = null;
cs.Create create = new cs.Create(new
ServiceInfoType(servicioB.Contract.Identifier));
remoteConstructor.Post(create);
yield return (Choice)create.ResponsePort;
CreateResponse createRsp = create.ResponsePort;
if (createRsp != null)
{
    servicioBService = createRsp.Service;
}
else
{
    LogError((Fault)create.ResponsePort);
    yield break;
}

_servicioBPort =
ServiceForwarder<servicioB.ServicioBOperations>(servicioBSe
rvice);
```

A continuación, se envía un mensaje Get (paso **4** de la *Figura 5.2.1-1*) al servicio socio, el servicio B. Para enviar este mensaje, se llama al método Get a través del

_serviceBPort declarado. Este construye y envía el mensaje Get al servicio. El Get devuelve por lo general el estado del servicio o fallo.

En el código mostrado a continuación se realiza de manera alternativa a la manera vista en apartados anteriores donde se hacía uso de Choice. Aquí, el estado del servicio es almacenado en una variable local en caso de éxito y en caso de fallo se loggea el error.

```
    servicioB.Get get;
    yield return _ servicioB
Port.Get(GetRequestType.Instance,out get);
    servicioB.ServicioBState state = get.ResponsePort;
    if (state == null)
    {
        LogError("Imposible obtener el estado del
ServicioB", (Fault)get.ResponsePort);
    }
```

En caso de tener éxito, es decir el estado sea distinto de null, se crea un nuevo estado del servicioA en el estado actual y se busca la definición del socio (paso **5** de la *Figura 5.2.1-1*).

```
    else // if (state != null)
    {
        ServicioAState initState = new ServicioAState();
        PartnerType partner = FindPartner("ServicioB");
        if (partner != null)
        {
            initState.Partner = partner.Service;
        }
        Replace replace = new Replace();
        replace.Body = initState;
        _mainPort.Post(replace);
    }
```

Teniendo ya el estado del servicioB, se concluye enviando un mensaje de suscripción al servicioB o loggeando el error(paso **6** de la *Figura 5.2.1-1*).

```
    servicioB.Subscribe subscribe;
    yield return _servicioBPort.Subscribe(_servicioBNotify,
out subscribe);
    Fault fault = subscribe.ResponsePort;
    if (fault != null)
    {
```

```
                LogError("Imposible de suscribir al servicioB",
            fault);
                }
}
```

Con todas estas modificaciones, ya se tiene el código listo para crear la aplicación distribuida.

Como último paso es necesario modificar el manifiesto del servicio A, que quedaría como se muestra a continuación:

```
<?xml version="1.0" ?>
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns:s="http://roboticus/2013/05/servicioa.html"
    xmlns:serviciob="http://roboticus/2013/05/serviciob.html"
    >
    <CreateServiceList>
        <ServiceRecordType>
<d:Contract>http://roboticus/2013/05/servicioa.html</d:Contract>
          <d:PartnerList>
            <d:Partner>
<d:Service>http://localhost:40000/directory</d:Service>
              <d:Name>s:Remote</d:Name>
            </d:Partner>
          </d:PartnerList>
        </ServiceRecordType>
        <ServiceRecordType>
      </CreateServiceList>
    </Manifest>
```

5.2.2 Ejecución de la aplicación distribuida

En caso de querer iniciar solo el servicio principal, debemos seguir los siguientes pasos.

En primer lugar hay que empaquetar[18] y distribuir al equipo correspondiente los servicios, en este caso el ServicioB a un equipo remoto.

Para esto se utiliza el Manifest Editor, o desde la dss command, ejecutamos:[19]
dssdeploy /p [options] Package.exe

Aplicado al ejemplo: dssdeploy /p /m:”servicioB.manifest.xml” serviciob.exe

Pudiendo añadir varios manifiestos en el mismo, añadiendo /m:”otro-manifiesto” a continuación del primero. U otros contenidos como .dll con el comando /d:”archivos-complementarios” después del manifiesto.

Una vez en el equipo remoto, se desempaqueta ejecutando directamente el paquete autoextraíble o con dssdeploy:

dssdeploy /u [options] Package.exe

Se puede utilizar el commando /t para especificar la carpeta de destino para desempaquetarlo mientras este no esté en un subdirectorio de la carpeta contenedora del paquete de instalación del DSS. Para nuestro ejemplo:

dssdeploy /u ServicioB.exe

Cuando se ha desempaquetado, podemos comprobar que todo está correctamente ejecutándolo desde la dss command prompt (ejecutando el script .cmd o con la instrucción dsshost /p:40000 /m:"ServicioB.manifest.xml). Comprobamos el nodo (en este mismo equipo obviamente) a través del navegador como se explica en 5.6.

Si hasta aquí todo va bien, pasamos al equipo local, donde en este caso se va a ejecutar ServicioA, y modificamos su manifiesto, indicando en la definición del servicio Remote la dirección IP del equipo remoto. Es decir, donde ahora se encuentra:

<d:Service>http://localhost:40000/directory</d:Service> , debemos cambiarlo a :

<d:Service>http://IP-Equipo-remoto:40000/directory</d:Service>

Utilizando 40000 o el número de puerto adjudicado, asegurándonos que está reservado.

Ahora, antes de iniciar el servicioA, hay que iniciar el nodo del equipo remoto. Con la dss command ejecutada como administrador, ejecutamos: dsshost /p:40000 . Importante no usar /t:40001.

Con este nodo abierto en el remoto, se procede a iniciar el nodo local también como administrador y ejecutar el servicioA.

Dsshost /p:50000 /m:"ServicioA.manifest.xml"

Todo esto tras acceder al directorio donde se encuentra el manifiesto. De esta forma, en el nodo Dss Remoto(40000), el del servicio B, se debe mostrar:

```
Service uri:  [http://IP-Equipo2:40000/servicioB]
```

Y en el local, donde se ejecuta el Servicio A(50000):

```
Service uri:  [http:// IP-Equipo1:50000/servicioA]
```

También podemos revisar los nodos a través del navegador como se explica en 5.6.

5.2.3 Comprobaciones de la red para distribuir nodos

Si al realizar lo explicado en el apartado anterior o no se conectan los servicios, no hay que preocuparse, hay que realizar unas comprobaciones en las configuraciones de los permisos para poder distribuir los servicios.[22]

Hay que comprobar que tanto el equipo remoto como el local se encuentran en la misma red(es decir, que solo se difieran en la última parte de la dirección IP) y ambas como domésticas. Que el equipo remoto tenga las configuraciones de firewall, es decir, desconectar desde el panel de control el firewall para las redes domésticas (recuerda activarlo al terminar para no dejar desprotegido el equipo) y se esté ejecutando como administrador.

En la siguiente imagen se ve cómo debe estar la red configurada:

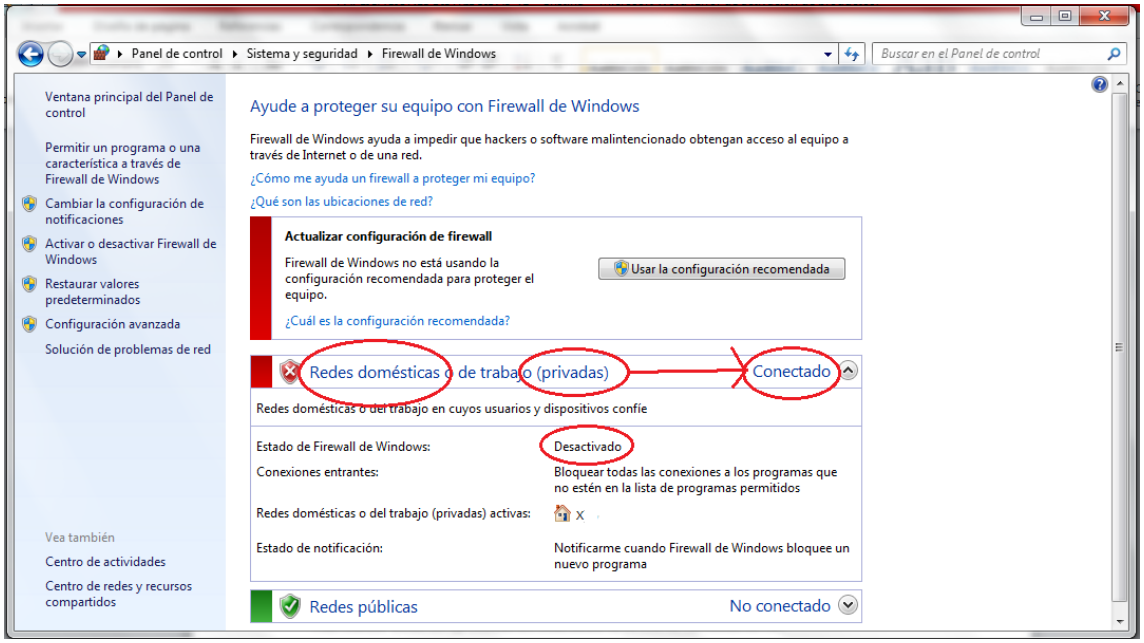


Figura 5.2.3-1 Configuración de la red

Y por último comprobar que el equipo remoto tiene la autentificación desactivada. Para comprobarlo insertamos en el navegador: http:/localhost:Num-puerto/Security/manager/Edit y ahí se cambia Authentication a “disabled”. Una vez hecho esto, se reinicia el nodo en el cual veremos en rojo “Security Alert!” como se observa en la *Figura 5.1.1-3.*

Tras realizar estas comprobaciones todo debería funcionar correctamente.

5.3 Aplicaciones distribuidas iniciando todos los servicios

Para este tipo de aplicaciones distribuidas, no es necesaria ninguna modificación en el código, pues vamos a iniciar manualmente cada uno de los socios, no son por tanto necesarios socios locales, ni ninguno de lo explicado en el apartado anterior. Lo que si será necesario, es crear una aplicación distribuida como tal, un manifiesto definiendo los distintos nodos, con los servicios que se ejecutarán en cada uno de ellos, esto se explica a continuación.

5.3.1 Creación del manifiesto de la aplicación distribuida

Para crear una aplicación distribuida con servicios existentes y con las debidas modificaciones explicadas anteriormente, utilizaremos en Microsoft DSS Manifest Editor[20].

Lo primero que debemos hacer para crear una aplicación distribuida, es comprobar que funciona la aplicación usando un solo manifiesto, es decir en un mismo nodo. En el ejemplo actual, eso se refiere a que podemos compilar y ejecutar la aplicación en nuestro propio equipo, tras hacer el manifiesto de forma habitual tal y como se explica en capítulos anteriores.

Para seguir, vamos a probar a ejecutarlo en nodos DSS distintos pero dentro del mismo equipo., y más tarde en equipos diferentes.

Creamos con el DSS Manifest Editor, una Nueva Aplicación Distribuida, (File-> Nuevo->Aplicación Distribuida).

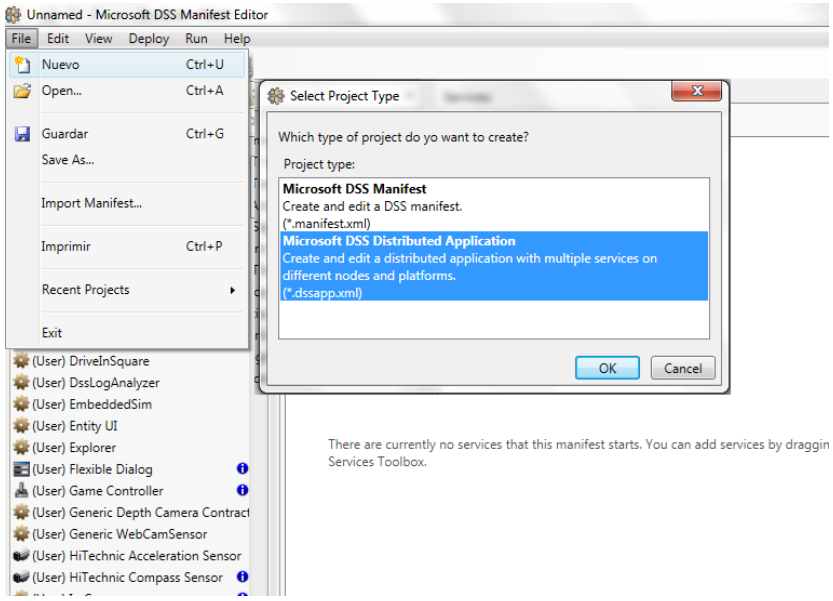


Figura 5.3.1-1 Captura de la herramienta Manifest Editor

Vemos como ahora podemos ver los distintos servicios unidos independientemente del número de nodos, en la pestaña “Application” o en los distintos nodos creados en la pestaña “Nodes”.

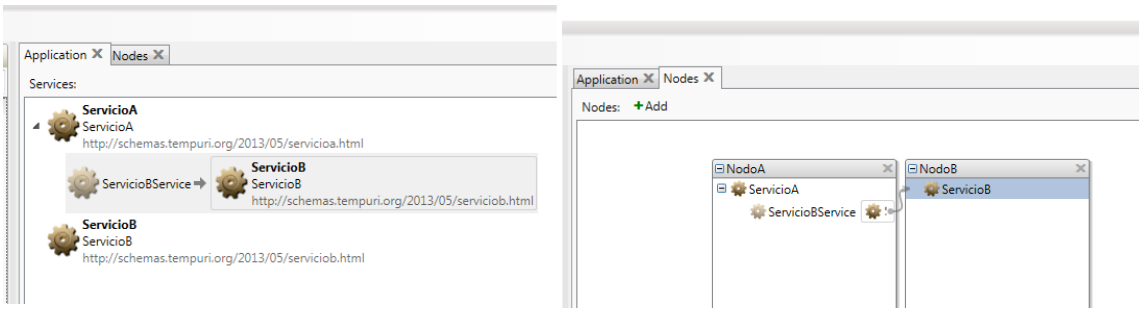


Figura 5.3.1-2 Captura de la herramienta Manifest Editor(Application a la izquierda y Nodes a la derecha.)

Inicialmente tenemos un solo nodo, para añadir más nodos (para crear una aplicación distribuida son necesarios al menos dos nodos) utilizar "Add->Microsoft Windows" (Importante, no hacer Microsoft Windows CE, pues no saldrá, es únicamente para aplicaciones especiales).

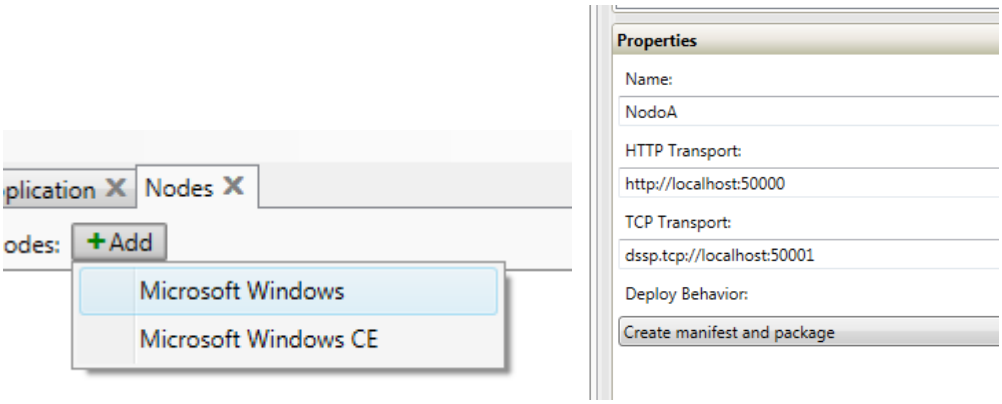


Figura 5.3.1-3 Captura de la herramienta Manifest Editor. Add Node(en la izquierda) y properties(en la derecha).

Como se aprecia en la imagen anterior, a la derecha, en la pestaña propiedades debemos especificar el nombre y los puertos que vaya a utilizar, especificando "localhost" o la dirección IP del quipo en el que se vaya a ejecutar. Más adelante se explicará cómo reservar los puertos que se vayan a utilizar, pues sin reservarlos, la aplicación no funcionará.

Una vez están creados los nodos, procedemos a insertar los servicios en los nodos deseados utilizando la pestaña "Services"(se observa en la parte izquierda de la figura *5.3.1*-3), tal y como hacemos al crear un manifiesto no distribuido.

Por último, debemos guardar el proyecto y crear los manifiestos y paquetes para ejecutar cada uno de los nodos. Para ello, el Manifest Editor nos proporciona una sencilla manera de hacerlo. En el menú "Deploy-> Create Deployment Package", este nos pedirá guardar el proyecto y seleccionar el directorio de salida para guardar los manifiestos y los paquetes de implementación.

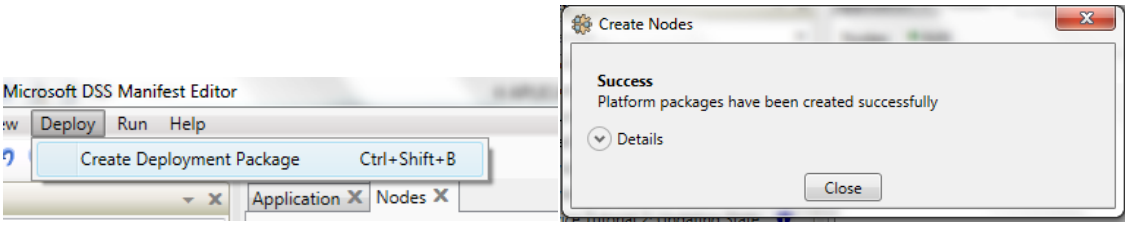


Figura 4.3.1-3 Captura de la herramienta Manifest Editor. Create Deploy package.

Es importante repetir este paso cada vez que se realicen cambios para asegurarnos que se reflejan en el manifiesto y los paquetes de implementación. También es importante, que el directorio de salida no sea el directorio de instalación del DSS ni ninguno de sus subdirectorios.

Cuando se han realizado con éxito los paquetes (Figura *5.3.1*-3 imagen izquierda), el directorio de salida contendrá para cada nodo tres archivos, el .exe que contiene todo lo necesario para ejecutar ese nodo, es el que usaremos para instalar el nodo en otro ordenador, .manifest.xml que especifica la lista de servicio que ejecutará el nodo, y .cmd un script que muestra como ejecutar el nodo manualmente.

Se distribuyen los paquetes de cada nodo a su equipo correspondiente (si es que se ejecutan en distintos equipos) según se ha definido al crear el paquete de implementación.

5.3.2 Ejecutar iniciando cada uno de los servicios necesarios

Una vez hayamos distribuidos los paquetes y reservado los puertos, ejecutamos de nuevo la dss command propmt como administrador. Nos trasladamos a través de los directorios hasta el que contiene el paquete de la aplicación distribuida (con la instrucción “cd directorio_destino”).

Una vez en el directorio, ejecutamos[21] los distintos ejecutables de los servicios y los distintos nodos en el orden que sea conveniente., cada uno desde la consola donde se desea ejecutar cada nodo. En este ejemplo, las instrucciones correspondientes a A serían en un equipo y las correspondientes a B en otro.

Por ejemplo, con el ejemplo que estamos explicando de dos nodos (véase Figura 5.1.1-1), cada uno con un servicio siendo A quien depende de B, sería:

1) A.exe
2) B.exe
3) Start B.cmd
4) Start A.cmd

Al hacer las dos primeras instrucciones nos avisará de la sobrescritura de los archivos, debemos aceptarlo. Y al realizar las dos últimas instrucciones se abrirá una nueva consola donde se empezará a mostrar los correspondientes mensajes del servicio. En

este ejemplo, en la consola del servicio A es la que se muestra en la *Figura 5.1.1-2*. La primera vez que se ejecute el nodo, al iniciar la consola del servicio, puede preguntarnos si permitimos acceso por estar ejecutando como administrador, debemos aceptarlo, y probablemente nos mostrará un error, esto es porque al ser la primera vez no contiene la carpeta Store y debe crearse.

5.4 Reserva de los puertos a utilizar

Antes de continuar, es muy importante confirmar que los puertos que se van a utilizar en cada uno de los equipos están reservados. Por defecto, un ordenador tiene reservado los puertos 50000 y 50001, en caso de utilizar solo estos no sería necesario reservarlos. En el caso de utilizar más de un nodo en un mismo equipo es necesario reservarlo. Supongamos que queremos ejecutar el Servicio B en el puerto 40000.

Para reservar los puertos hay que seguir los siguientes pasos:

1) Puesto que se ha aumentado la seguridad en esta versión, se ha desactivado la comunicación en red entre los nodos de forma predeterminada. Así pues, lo primero que debemos hacer es habilitarla manualmente en las propiedades de DSS Application Configuration File (en el fichero DssHost.exe.config o DssHost32.exe.config) que podemos encontrar a través del menú de Inicio de Windows. Debemos permitir acceso remoto sin garantía, para ello cambiamos el valor de "AllowUnsecureRemoteAccess " a true. Esto permanecerá con el nuevo valor de forma indefinida. Recuerda repetir este paso si re-instalas el programa.
   Los paquetes de implementación contendrán este la configuración de nodos (dsshost.exe.config) con las propiedades del ordenador que se usó para crear los paquetes. Este incluye los ajustes de seguridad que se hayan hecho en el fichero. Si se quiere cambiar este fichero para el paquete creado, esto se puede hacer antes de generarlo o después de haber implementado los paquetes.
2) Reservar los puertos con httpReserve. Esto se hace a través de la consola, no en las propiedades del nodo.
   Debemos ejecutar la consola (dss command propmt) como administrador, pues los puertos deben ser reservados por el administrador antes de usarlos.
   Para ejecutar la consola como administrados tenemos varias opciones, abrir la consola normal y cambio en las propiedades: Start in: %HOMEDRIVE%%HOMEPATH%
   Así en la consola debería poner C:\Windows\System32\ y no C:\Users\username , o simplemente Administrador en la cabecera. Así ejecutamos con los privilegios de administrador.

Una vez la consola está ejecutándose como administrador, escribimos:

Httpreserve /p:<número de puerto> /u:<nombre de usuario> , como por ejemplo Httpreserve /p: 40000 /u:Cris

5.5 Comprobación de los nodos

Para comprobar[21] las conexiones y los mensajes de cada uno de los nodos, hay que acceder a través del navegador, a *http://localhost:numero-puerto/nombre-servicio*, en este caso por ejemplo *http://localhost:50000/servicioA* o *http://localhost:* 40000*/servicioB* ahí veremos el estado de cada uno de ellos. O podemos acceder al nodo directamente *http://localhost:numero-puerto* y ver la consola, o cada uno de los servicios conectados.

Esta es una buena manera de comprobar los posibles errores, o simplemente curiosear los servicios de cada nodo.

5.6 Aplicaciones distribuidas con simulación

Es importante resaltar, que no es posible crear paquetes de implementación con el entorno de simulación en ellos.

DssDeploy no se puede utilizar para empaquetar el simulador. Cualquier comando DssDeploy que incluye un manifiesto que hace referencia al motor de simulación fallará.

En el caso de la simulación, es necesario tener instalado MSRS junto con DirectX, XNA y Ageia en el equipo de destino antes de implementar su servicio a la misma.

Esto se debe a que la simulación tiene otras dependencias a parte de las dlls del CCR y DSS. Por ejemplo, el entorno de simulación requiere la instalación de DirectX junto a XNA, no vale simplemente con copiar el dll del XNA, si no que se requiere tenerlo instalado correctamente en el sistema.

Además de las dependencias de código, probablemente hay otras dependencias para su proyecto que DssDeploy no recoge de forma automática. Por ejemplo, incluso las escenas más básicas probablemente necesitarán una malla cielo (SkyDome.dds) y textura (sky_diff.dds, SkyDome.dds) y la textura del suelo (por defecto es 03RamieSc.dds). Si el servicio utiliza las otras mallas y texturas, se tendrán que incluir explícitamente en el paquete utilizando la opción de línea de comandos / d.

5.7 Ejemplo

Tras realizar detenidamente el ejemplo expuesto en los apartados anteriores, ya sabemos las nociones básicas necesarias para realizar un ejemplo un poco más

complejo. Así pues, se va a definir un nuevo ejemplo, vamos a definir un Centro de Control de alarmas, para lo cual se dispondrá de un nuevo sistema de módulos.

Nuestro sistema de módulos será, como se muestra en la siguiente figura. Con este esquema podríamos realizar simulación de sistemas que representen cualquier tipo de centro de control, aprovechando la estructura de módulos definida al igual que en el capítulo 2, se aprovecha el mismo sistema de módulos para todos los sistemas.

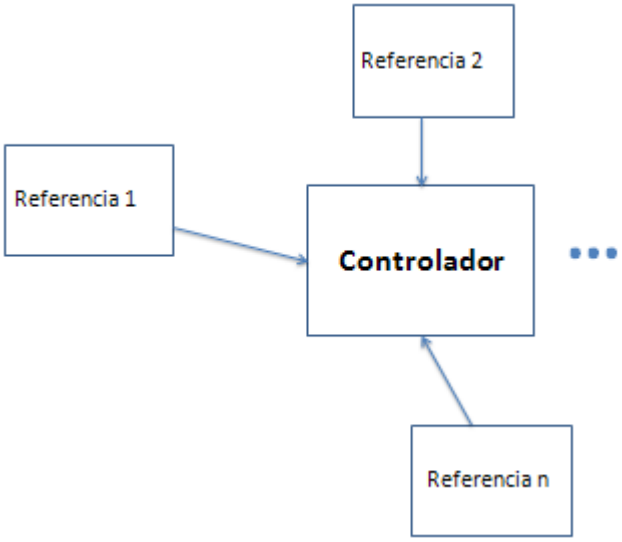


Figura 5.8-1 Esquema de los nodos

Como vemos, tenemos dos tipos de servicios, un servicio Controlador que es el que gestiona el control del resto de módulos. Define tantos socios como Referencias se quieren tener en el centro de control. Se deben definir de la forma explicada en el apartado 5.2.1.

Y tenemos un servicio referencia, creado de forma normal, como en capítulos anteriores, con una interfaz gráfica sencilla, con simplemente un botón para poder enviar la alarma. Para poder tener varias instancias de referencias actuando del mismo modo, debemos modificar AllowMultipleInstances = true, en la declaración del servicio (en el .cs)

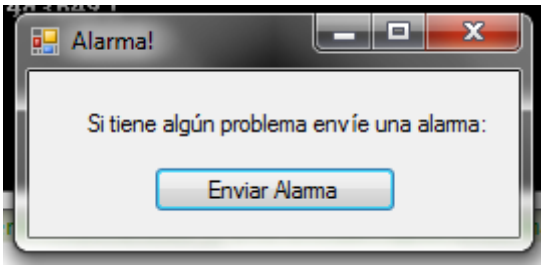


Figura 5.8-2 Interfaz de referencia

De esta forma, ya tenemos los servicios creados, los socios definidos y suscritos. Es necesario definir en el manifiesto del controlador las partners con la dirección IP de cada uno de sus socios.

Suponiendo que ejecutamos el controlador en el equipo local y se distribuyen los nodos donde se ejecutan referencia, se deben empaquetar el servicio Referencia como se explica en el apartado 5.2.1.

Con el ejemplo que estamos creando se distingue claramente la ejecución distribuida de los nodos. Pues avisando de la alarma en un equipo, se ve reflejado el aviso en el equipo que contiene el controlador.

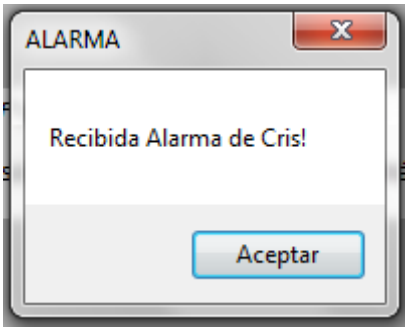


Figura 5.8-3 Salida del controlador

Como salida por ambas consolas, obtenemos las siguientes imágenes:

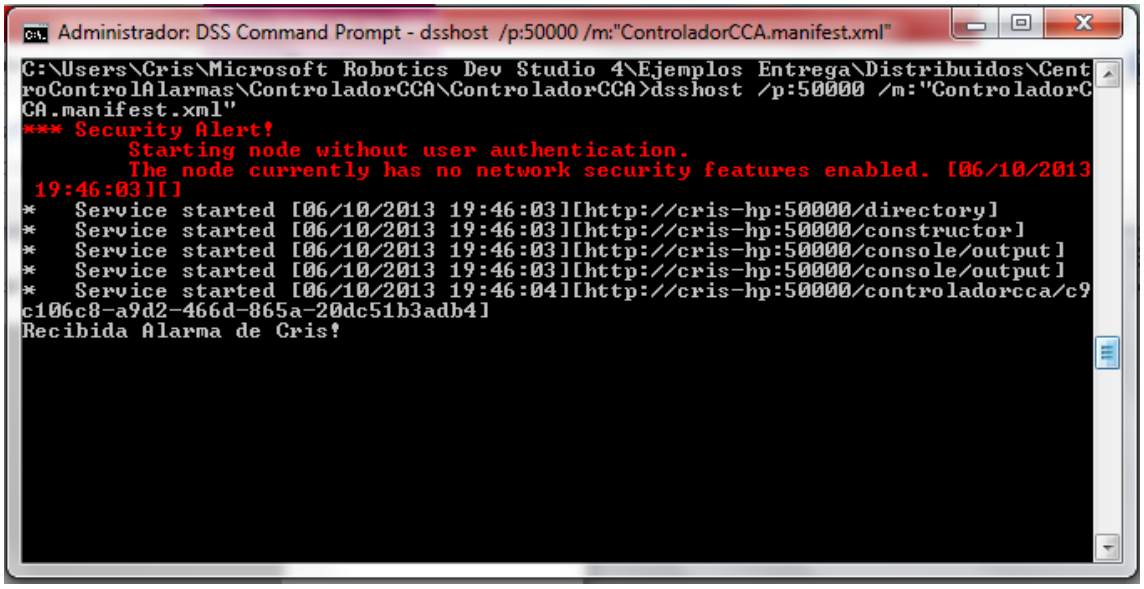


Figura 5.8-4 Consola Controlador

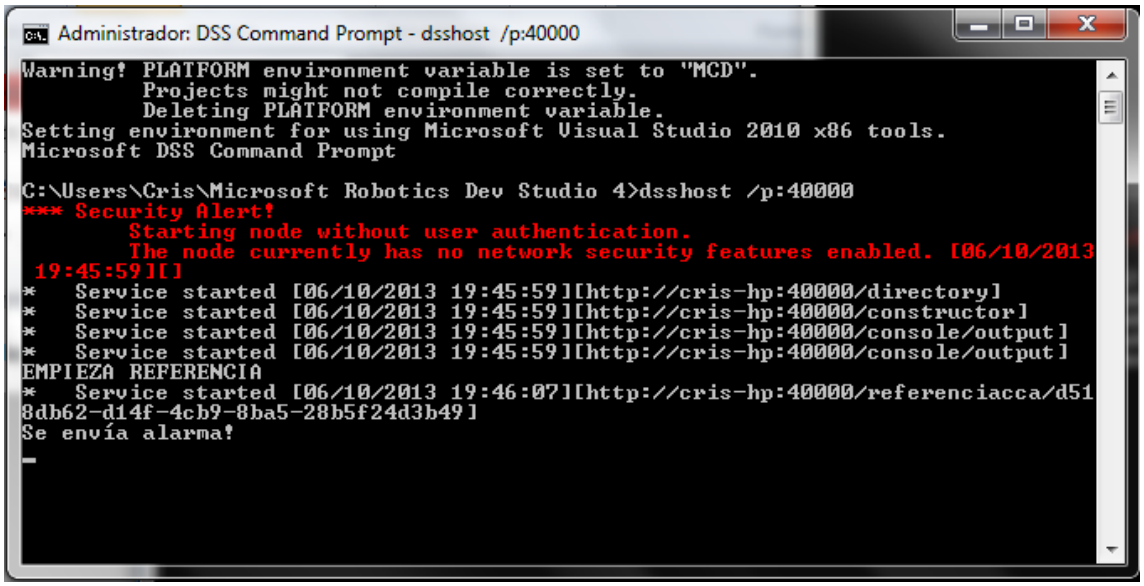


Figura 5.8-5 Consola Referencia

5.8 Conclusiones

Tras haber aprendido a realizar las aplicaciones distribuidas, y aun pareciendo complejo a primera vista, es un mecanismo muy útil a la hora de realizar grandes aplicaciones, aplicaciones con necesidad de tener una gran potencia de cálculo, aplicaciones con necesidad de traspasar un servicio a un hardware determinado, o aplicaciones en las que es necesario tener el software en distintos equipos. Así pues, se considera una de las grandes ventajas del uso de este software para el proyecto, pues con unas modificaciones de código, se añade una funcionalidad muy importante.

6. Conclusiones

Con el proyecto ya finalizado, se pueden concluir varias cosas. Una de ellas todo lo que hemos aprendido, las herramientas nuevas que en un principio nos eran totalmente desconocidas, pero que tras todo el trabajo y esfuerzo invertido ya podemos manejar con soltura. Los conceptos nuevos que hemos añadido a nuestros conocimientos, igual no tan relacionados directamente con el tema del proyecto, como la unión de nodos y las configuraciones de seguridad de red.

Pero principalmente, concluimos que el sistema de módulos diseñado ha resultado ser tan eficaz como planeamos al inicio del proyecto. A lo largo de los capítulos hemos diseñado varios ejemplos de distintos tipos de sistemas, siguiendo una misma plantilla de módulos, demostrando así, como es igual de eficaz para cada uno de ellos, siendo necesario cambiar simplemente el comportamiento de estos, o alguno de estos, pero no siendo necesario modificar las relaciones de módulos, es decir, la declaración de socios y la suscripción de sus puertos. Habiendo comprobado esto, se concluye que un usuario sin conocimiento de cómo realizar relaciones de socios, ni relacionar nodos, puede ser capaz de ejecutar el sistema que desee siempre que tenga una buena plantilla como la descrita aquí, y por supuesto, nociones de programación para realizar los cambios de código que necesite para obtener los resultados de su servicio.

En cuanto a las aplicaciones distribuida, no solo hemos sacado como conclusión su efectividad como propiedad de la aplicación, si no que añadiendo todas sus ventajas a las ventajas del propio proyecto, es decir, al uso de una plataforma reconfigurable de módulos, en una plantilla para aplicaciones distribuidas ya estarían realizadas las modificaciones de código necesarias para ejecutarla en diferentes nodos, por lo que el usuario podría simular sus sistemas en diferentes equipos sin necesidad de tener conocimiento sobre cómo distribuir una aplicación. Para obtener el conocimiento necesario para la configuración de la red y la modificación de manifiestos para especificar las direcciones concretas de los nodos, se dispone en el presente trabajo de una guía explicativa de todos los pasos necesarios a seguir.

7. Apéndices

7.1 Apéndice 1: Instalación de MRDS

Para la instalación[23] del Microsoft Robotics Developer Studio tenemos tres requisitos. Uno de ellos es que debe instalarse con Windows 7. Además, tiene que usarse con alguna de las ediciones del Visual Studio 2010. Y por último, el ordenador debe tener una tarjeta gráfica compatible con gráficos DirectX 9 con Shader 2.0 Vertex y Pixel Modelo o superior. La mayoría de los sistemas vendidos desde 2006 con gráficos en 3D cumple con estos requisitos. Hay algunos sistemas de gama baja con chips de gráficos integrados, como la familia de chipsets Intel 865G, que no cumplen con estos requisitos.

Una vez comprobado que cumplimos los requisitos, debemos descargarnos una de las versiones de Visual Studio 2010 con C#. Es un entorno de desarrollo integrado para sistemas operativos Windows. Con él, podemos programar en varios lenguajes de programación como C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET. Debido a que C# es el lenguaje de programación más adecuado para la implementación de nuestro proyecto, hemos utilizado dicha herramienta que es la que más nos facilita su desarrollo.

Con Visual Studio admitimos numerosas herramientas que hacen la implementación en C# mucho más sencilla que con cualquier otro editor. Posee un editor de código completo, plantillas de proyecto, asistentes para código, un depurador de fácil manejo y eficaz, además de otras muchas herramientas. También posee una biblioteca de clases .NET Framework que ofrece acceso a un número elevado de servicios y a otras clases que pueden resultar útiles y que pueden resolvernos numerosos problemas. Al combinar .NET Framework con C#, podemos crear aplicaciones para Windows, servicios Web, herramientas para bases de datos y mucho más.

Así pues, si planeamos escribir nuestros propios servicios, debemos instalarlo, pues este entorno de desarrollo contiene todos los componentes necesarios para crear servicios MRDS.

A continuación, nos descargamos e instalamos el Kinect para Windows SDK V1 y Silverlight 4.0 SDK. Una vez tenemos instalados estos tres programas, procedemos a descargar e instalar también de forma gratuita el MRDS.

Cuando la instalación se ha completado correctamente, tenemos que ejecutar todos los ejemplos. En el menú Inicio, en la carpeta de Microsoft Robotics Developer Studio 4, ejecutar “Build All Samples”.

7.2 Apéndice 2: Contenido adjunto

7.2.1 Contenido

En el cd que viene adjunto a esta memoria, encontramos en primer lugar, una carpeta con el nombre “Instalación” con todo lo necesario para instalar el programa usado durante el proyecto MRDS (explicación en 7.1 Apéndice 1).

Encontramos también una carpeta con el nombre “Ejemplos Entrega”, con todos los ejemplos citados a lo largo del trabajo, se encuentran situados en distintas carpetas, a saber:

- Quatrirrotor: contiene los cuatro servicios creados.
- SistemaDiscreto: Contiene los cuatro servicios creados.
- Sistema Continuo: Contiene los seis servicios creados.
- Sistema DEVS Atómico: Contiene los tres servicios creados.
- Sistema DEVS Acoplado: Contiene los dos servicios creados.
- Ditribuidos: Contiene los ejemplos creados como aplicaciones distribuidas:
    - ServicioA
    - ServicioB
    - CentroControlAlarmas: con los dos servicios creados.

Y por último, encontramos una carpeta con los ejecutables de cada uno de los ejemplos que no contienen simulación, es decir, todos exceptuando el quatrirrotor que habrá que ejecutarlo de la forma que se explica en el siguiente apartado, pues es la única forma posible.

7.2.2 Ejecución de los ejecutables.

Para los ejecutables que se encuentran en el contenidos del cd, debemos ejecutar una dss command como administrador, movernos hasta el directorio contenedor del ejecutable, en nuestro caso /Ejemplos entrega/Ejecutables, a través del comando cd.

Una vez en el directorio adecuado, hacer: nombreEjemplo.exe. Se muestra ena ventana para elegir el directorio de salida. Tras seleccionarlo, se ha descomprimido el paquete creando en el directorio que hayamos elegido la misma ruta que se hubiese guardado el proyecto. En este caso, si por ejemplo desempaquetamos ServicioA.exe, y dejamos la carpeta Ejecutables como salida, se creará en Ejecutables una carpeta bin con las dll desempaquetadas, y una carpeta Ejemplos

Entrega/Distribuidos/ServicioA/ServicioA donde se encuentra el manifiesto que podemos ejecutar de forma habitual:

Ddshost /p:50000 /m:"servicioA.manifest.xml"

7.2.3 Migración y ejecución de los ejemplos migrados.

Para poder ver el código realizado a la vez que poder ejecutarlos, es necesario "migrar" [19]los proyectos que vienen en Ejemplos entrega. Para esto, tras tener instalado todo lo necesario (tras seguir el apéndice 1), copiamos la carpeta Ejemplos entrega en el directorio que se ha creado Microsoft Robotics Developer Studio 4.

Ejecutamos la dss command (se puede acceder a ella a través del menú de Inicio) como Administrador. Nos situamos en el directorio de Ejemplos Entrega (cd Ejemplos Entrega), y Migramos cada uno de los proyectos:

Dssprojectmigration /b- NombreCarpetaProyecto

De esta forma, ya hemos migrado el proyecto para que las rutas coincidan con el equipo actual, no con el equipo que creo el proyecto.

Una vez migrados todos los proyectos, ya se pueden ejecutar, o bien abriendo el Visual Studio y ejecutando directamente desde ahí, o por la dss commnd como se explica a continuación.

Se ejecuta la dss command como Administrador, y cambiamos de directorio hasta el del proyecto a ejecutar, por ejemplo Ejemplos entrega/SistemaDiscreto. Para ejecutar el proyecto completo, es necesario haber creado la solución y las dll de cada uno de sus módulos, en este caso, ReferenciaSD, ControladorSD, SensorSD y PlantaSD. Para ello, para cada uno de ellos, en la dss command nos situamos en su directorio, por ejemplo Ejemplos Entrega/SistemaDiscreo/ReferenciaSD y ejecutamos :

Msbuild referenciaSD.sln

A continuación, hacemos de nuevo cd referenciaSD, y ejecutamos el programa:

Ddshost /p:50000 /m:"referenciasd.manifest.xml"

De esta forma, se ha creado la solución y la dll. Se habrá mostrado en la consola un mensaje en amarillo como el que se muestra en la siguiente figura:

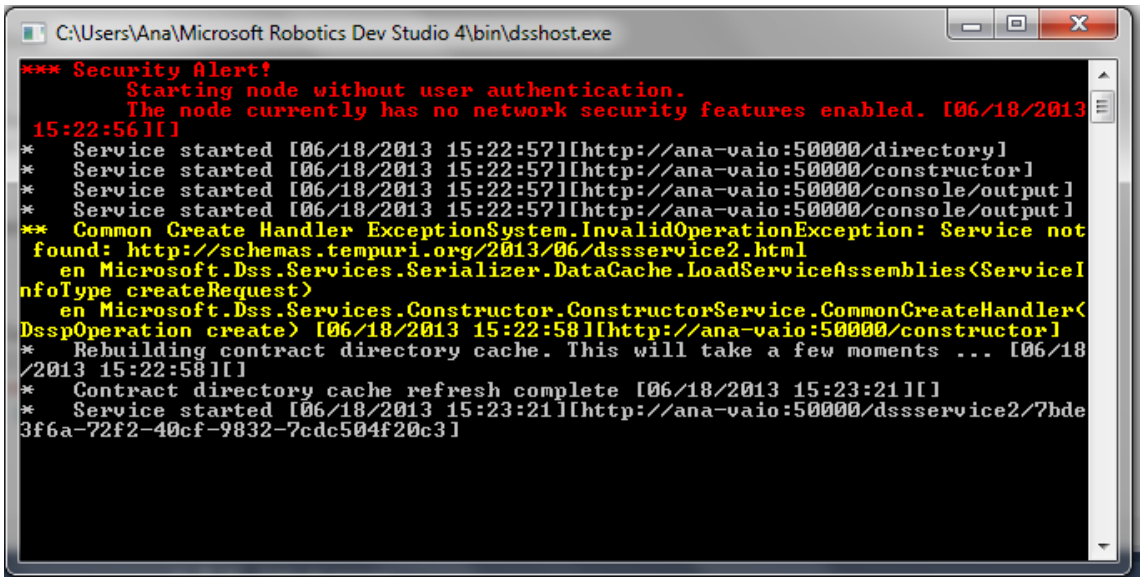


Figura 7.2.3 -1 Consola al crearse la solución y dll

8. Bibliografía

[1] El libro en el que más nos hemos apoyado, en general para todo el proyecto:

Professional Microsoft Robotics Developer Studio;
Autores: Kyle Johns, Trevor Taylor.
Editorial: Wrox
Fecha edición: Mayo 2008

Recursos utilizados para Introducción (Capitulo 1):

[2] Herramientas de simulación:

Open Dynamic Engine: http://www.ode.org/

Cafu Engine: http://www.cafu.de/
http://en.wikipedia.org/wiki/Cafu_Engine

AGX Multiphysics: http://en.wikipedia.org/wiki/AGX_Multiphysics
http://www.algoryx.se/agx

NVidia PhysX: http://www.nvidia.es/

JigLibX PhysX Library: http://jiglibx.codeplex.com/

Bullet: http://bulletphysics.org/wordpress/

MATLAB /Simulink: http://www.mathworks.es/products/simulink/

Seamless 3D: http://www.seamless3d.com/

VisSim: http://www.vissim.com/
http://es.wikipedia.org/wiki/VISSIM

SimApp: http://www.simapp.com/

AnyLogic: http://www.anylogic.com/
http://es.wikipedia.org/wiki/AnyLogic

[3] Explicaciones sobre MRDS para introducción de cada una de sus partes:

http://msdn.microsoft.com/en-us/library/dd939239.aspx

Recursos utilizados para MRDS (Capitulo 2):

[4] Explicaciones sobre MRDS:

http://www.microsoft.com/robotics/

[5] Explicación sobre CCR:

http://msdn.microsoft.com/en-us/library/bb905470.aspx

[6] Explicaciones sobre DSS:

http://msdn.microsoft.com/en-us/library/bb905471.aspx

[7] Explicación sobre servicios:

http://msdn.microsoft.com/en-us/library/bb483050.aspx

[8] Explicación sobre tiempos:

http://msdn.microsoft.com/es-es/library/system.timespan.aspx
http://msdn.microsoft.com/es-es/library/system.datetime.aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-6
http://msdn.microsoft.com/es-es/library/system.timers.timer(v=vs.80).aspx

Recursos utilizados para Sistemas (Capitulo 3):

[9] Actualización del estado

http://msdn.microsoft.com/en-us/library/bb483062.aspx

[10] Suscripciones de unos servicios a otros

http://msdn.microsoft.com/en-us/library/bb483060.aspx

http://msdn.microsoft.com/en-us/library/bb483061.aspx

[11] Explicaciones sobre servicios y conexiones

http://msdn.microsoft.com/en-us/library/bb483059.aspx

[12] Tutoriales para WinForms y creación de gráficas

http://msdn.microsoft.com/en-us/library/dd456632.aspx

[13] Explicación para DEVS:

http://xsy-csharp.sourceforge.net/DEVSsharp/
http://www.cs.gsu.edu/DEVSTutorial/

Recursos utilizados para Simulación (Capitulo 4):

[14] La clase y el espacio de nombres del motor de simulación:

http://msdn.microsoft.com/en-us/library/bb843796.aspx

[15] Tutoriales y ejemplos del VSE:

http://msdn.microsoft.com/en-us/library/bb896718.aspx

[16] Explicaciones sobre las entidades y sus propiedades:

http://www.funnyrobotics.com/2010/10/microsoft-visual-simulation-environment.html

Recursos utilizados para Aplicaciones Distribuidas (Capitulo 5):

[17] Los tutoriales de servicios, en especial el 5, el 7 y el 11:

http://msdn.microsoft.com/en-us/library/dd145263.aspx

[18] Explicaciones para el uso del empaquetamiento de servicios individuales.

http://msdn.microsoft.com/en-us/library/bb483012(v=MSDN.10).aspx

[19] Explicación de las herramientas DSS, entre ellas destaca el uso del DssDeploy, DssHost y DssProjectMigration. Para distribuir los servicios, ejecutarlos y migrar proyectos respectivamente.

http://msdn.microsoft.com/en-us/library/cc998572.aspx

[20] Explicación del uso del Dss Manifest Editor, herramienta incorporada con MRDS, muy útil para crear los manifiestos de los servicios.

http://msdn.microsoft.com/en-us/library/bb643213.aspx

[21] Como ejecutar servicios y explorarlos con el navegador:

http://msdn.microsoft.com/en-us/library/bb483055(v=MSDN.10).aspx

[22] Los foros de MSDN, en especial una explicación muy detallada sobre los fallos que se podían encontrar:

http://social.msdn.microsoft.com/Forums/en-US/roboticsdss/thread/9fc6999d-66c9-4ab2-8a7c-58d0c4516c59

[23] Se pueden obtener todas las descargas necesarias en

http://www.microsoft.com/en-us/download/details.aspx?id=29081